Liesert

PEEKS & POKES

zum Commodore 64

Ein DATA BECKER Buch

ISBN 3 - 89011 - 032 - 0

3. erweiterte und überarbeitete Auflage

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.



Vorwort

Sie kennen das Problem: Die mitgelieferte Anleitung des Commodore 64 haben Sie durchgelesen. Schon nach kurzer Zeit wollen Sie mehr wissen, als die unbestreitbar ungenügende CBM-Publikation hergibt. Sie fragen sich vielleicht, wie man die groß angekündigte Kollisionskontrolle bei Sprites durchführt, wie man hochauflösende Grafik erzeugt, oder wie man zwei Tasten gleichzeitig abfragen kann. Im Anhang befindet sich zwar eine Liste mit Einzelheiten aus der Zeropage, aber:

1. Was ist eine Zeropage überhaupt?

und

2. Wie mache ich sie mir zunutze?

Wenn dies Ihre Probleme sind, dann halten Sie das richtige Buch in Händen.

Wir werden zusammen eine Reise durch Speicher und Betriebssystem des 64ers unternehmen - wie, Sie wissen nicht, was ein Betriebssystem ist? Macht nichts, auch das wird erklärt.

Zu diesem Zweck besteht das Buch aus drei Teilen. Im ersten Abschnitt schaffen wir die Grundlagen für die dann folgenden Tricks (Abschnitt 2). Dazu gehört die Erläuterung der BASIC-Befehle PEEK, POKE und dergleichen mehr. Wenn Sie nach diesem Abschnitt dann die Programmierung und Funktionsweise Ihres Rechners besser verstehen, folgen eine Menge Tricks, die alle vom BASIC aus funktionieren. Sie benötigen also keine Maschinensprachekenntnisse, um in Zukunft komfortabler programmieren zu können.

Jedem Abschnitt mit Tricks ist eine kurze Zusammenfassung nachgestellt, damit man beim späteren Nachschlagen nicht unbedingt alle Erläuterungen mitlesen muß.

Wenn Sie sich genau an die Beschreibungen halten, kann ich Ihnen garantieren, daß alle Tricks und Programme funktionieren.

Apropos Maschinensprache: Im 3. Abschnitt finden Sie ein Simulationsprogramm für einen Miniprozessor und eine kleine Einführung in die Anfänge der Maschinensprache, die Ihnen den Einstieg in weitere Lektüre erleichtern soll.

Besitzern des VC-20 sei gesagt, daß fast alles, was sich auf die Zeropage bezieht, prinzipiell auch auf dem kleinen Bruder des 64ers laufen kann, wenn auch mit kleinen Änderungen. Hier hilft vielleicht ein kleiner Blick in die bekannten DATA-BECKER-Bücher.

Mir bleibt nur noch, Ihnen viel Spaß bei der Nutzung der neuen Möglichkeiten in der Programmierung Ihres CBM zu wünschen.

Hans Joachim Liesert
Münster, im Mai 1984

Vorwort zur überarbeiteten Ausgabe

Seit Erscheinen der ersten Ausgabe der "Peeks & Pokes" haben mich viele Leserbriefe mit Hinweisen und Tips erreicht. Natürlich haben Sie auch die unvermeidlichen Fehler entdeckt, und ich hoffe, daß es mir diesmal gelungen ist, letztere auszuschalten. Allen, die daran Anteil hatten, sei hiermit mein herzlichster Dank ausgesprochen.

Für die Freaks und POKE-Fetischisten wurde außerdem der Anhang (insbesondere der Speicherbelegungsplan) erweitert. Ich hoffe, daß Ihnen dies bei der Erstellung eigener Programme eine Hilfe ist.

Hans Joachim Liesert
Münster, im Februar 1986

Achtung: **Aus drucktechnischen Gründen wird in diesem Buch der Hochpfeil als ^ dargestellt.**



Inhaltsverzeichnis

1. Die Arbeitsweise des Rechners

In den folgenden Abschnitten lernen Sie den 64er und seine Funktionsweise kennen. Diejenigen, die schon einigermaßen sattelfest in der Computertechnik sind, können getrost weiterblättern. Die "Supercracks" unter Ihnen mögen mir etwaige Vereinfachungen verzeihen, die ich des besseren Verständnisses wegen gemacht habe.

1.1. Der Mikroprozessor

Die Freaks und Experten reden meist nur noch von Typenbezeichnungen wie 6510, 68000 oder Z-80, wenn sich ihre Fachsimpelei wieder einmal um den Mikroprozessor ihres Leib- und Magenrechners dreht. Doch was tut so ein Prozessor überhaupt? Nun, der Name deutet es an: dieser Baustein auf der Platine übernimmt den eigentlichen Datenverarbeitungsprozeß, d.h. er holt sich Daten vom Speicher herein, verarbeitet sie (z.B. durch Addition oder Subtraktion) und liefert sie wieder an den Speicher aus. Als "Lieferwagen" für den Transport der Daten dient der sogenannte Datenbus. Das sind einfach 8 Leiterbahnen, die zwischen Mikroprozessor und Speicher verlegt sind. Jede dieser Leiterbahnen transportiert jeweils ein Bit; 8 Bits ergeben ein Byte.

1 Bit ist die kleinste Informationsmenge, die es gibt: 0 oder 1, "JA" oder "NEIN", bzw. Strom aus oder Strom an. Dies ist elektronisch viel leichter zu bearbeiten als z.B. 10 verschieden große Ströme für die Ziffern 0 bis 9.

Der 6510 (das ist der Mikroprozessor, der in Ihrem C-64 Dienst tut) gehört zur Klasse der 8-Bit-Prozessoren, weil er jeweils 8 Bits auf einen Schlag bearbeitet.

1 Byte entspricht einem Buchstaben oder den Zahlen 0 bis 255. Das RAM Ihres C-64 enthält 65536 Bytes. Woher weiß aber der Speicherbaustein, in welche dieser Speicherzellen das Byte, das gerade per Datenbus angeliefert wird, kommen soll? Ganz ein-

fach: der Mikroprozessor schickt auf dem Adressbus die Adresse der gewünschten Zelle sozusagen als Lieferschein mit. Der Adressbus hat 16 Leitungen, weil 216 = 65536 verschiedene Ziele möglich sind. Will man eine Speicheradresse im Speicher aufbewahren, braucht man dazu also 2 Byte.

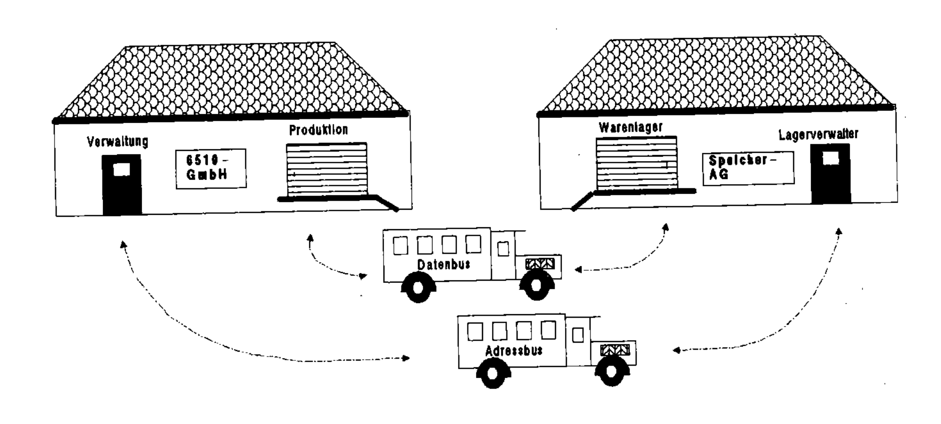


Abb. 1: So greift der Prozessor auf den Speicher zu

1.2. Was ist ein Betriebssystem?

Wenn Sie sich mit einschlägiger Computerliteratur befassen, so werden Sie oft auf Wörter wie "Betriebssystem", "Interpreter" oder (besonders beim CBM-64) "Interrupt" stoßen.

Was hat es also mit diesen Begriffen auf sich? Die unscheinbare "Plastikkiste", die sich da auf Ihrem Schreibtisch breit macht, erweckt leider viel zu leicht den Eindruck, es handele sich dabei um eine Maschine, die, solange alle Bauteile nur genug Saft bekommen, einfach nicht anders kann als zu funktionieren. Dem ist leider nicht so. Bevor ein Mikroprozessor irgendetwas tun kann (und sei es nur das Warten auf eine Eingabe), muß er ein Programm (in Maschinensprache) haben, das ihm sagt, was zu machen ist. Das fängt schon beim Einschalten des Rechners an.

Just in diesem Moment nämlich schnappt sich der Mikroprozessor die beiden allerletzten Bytes im Speicher. Diese stellen eine Adresse dar, die besagt, wo das Startprogramm zu finden ist (es steht übrigens im ROM). Dieses Programm sorgt dafür, daß der Speicher aufgeräumt, die Einschaltmeldung auf den Bildschirm gebracht und vieles mehr getan wird. Dann erst wird der BASIC-Interpreter gestartet.

Die Funktion des Interpreters (engl. Übersetzer) ist dem Namen leicht zu entnehmen. Es handelt sich hier einfach um das Programm, das die BASIC-Anweisungen, die Sie eingeben, für den Computer übersetzt.

Der 64er und alle anderen Mikrocomputer können nämlich eigentlich nur ihre spezielle Maschinensprache verstehen. Erst der BASIC-Interpreter, der in einem ROM gespeichert ist, ist in der Lage, Programmzeilen aus dem Speicher zu holen und diese abzuarbeiten. Wenn das BASIC im Direktmodus abläuft, holt es die Anweisungen nicht aus dem Programmspeicher, sondern aus dem BASIC-Eingabepuffer.

Dieser Eingabepuffer stellt eine Art "Übergabebereich für Tastendrucke" dar, d.h. das Betriebssystem (ein weiteres Programm im ROM), das für die Abfrage der Tastatur, die Erzeugung des Cursors und die Bedienung der Peripheriegeräte zuständig ist, teilt dem BASIC hier mit, was der Anwender "draußen vor der Tastatur" eingegeben hat.

Damit sind wir auch schon bei den Aufgaben des Betriebssystems. Hierbei handelt es sich nicht um einen Teil der Hardware. Im Gegenteil, auch das Betriebssystem ist ein Programm in Maschinensprache.

Immer, wenn in einem Programm Dinge getan werden sollen, die mit der Ein- und Ausgabe von Daten zu tun haben(z.B. Zeichen drucken, Tastatur abfragen, Floppy ansprechen), tritt es in Aktion. Man muß also nicht selbst die Tastatur ansprechen, sondern sagt einfach dem Betriebssystem, es möge doch bitte-

schön den nächsten Tastendruck in dieser und jener Speicherzelle vermelden.

Wie Sie von der BASIC-Programmierung her wissen, kann ein Computer (Multiprozessorsysteme ausgenommen) immer nur ein Programm ausführen. Interpreter und Betriebssystem sind aber zwei getrennte Programme, die zur Erledigung bestimmter Aufgaben simultan ablaufen müssen. Wie wird dieses Problem gemeistert?

Die einfachste Möglichkeit, zwei Programme fast gleichzeitig ablaufen zu lassen, ist der gegenseitige Aufruf. Immer wenn das BASIC mit einem Teil seiner Arbeit fertig ist, schaltet es das Betriebssystem ein und umgekehrt. Dies geschieht zum Beispiel, wenn auf Peripheriegeräte zugegriffen werden soll. Das BASIC stellt lediglich die Informationen zur Verfügung, die das Betriebssystem dann zum Gerät schickt. Dies beinhaltet aber auch, daß z.B. die Tastatur nur dann abgefragt wird, wenn das Betriebssystem gerade läuft. Nun soll aber während des Programmlaufs zumindest die RUN/STOP-Taste eine sinnvolle Wirkung zeigen. Um dieses Problem zu lösen, erfanden die Computerhersteller den INTERRUPT (engl. Unterbrechung). Jede $^1/_{60}$ Sekunde unterbricht der Prozessor das gerade laufende Maschinenprogramm (ob BASIC oder Betriebssystem) und springt in die Unterprogramme für Tastaturabfrage und ähnliche Dinge. "Entdeckt" der Rechner dabei einen Druck auf die RUN/STOP-Taste, so wird das gerade laufende BASIC-Programm abgebrochen. Sollte eine andere Taste gedrückt worden sein, so wird dies im Tastaturpuffer (übrigens eine sehr nützliche Einrichtung) gespeichert.

Für den Anwender scheint es, als ob die Tastatur ständig abgefragt würde, da selbst der schnellste Tipper wohl kaum mehr als 15 Zeichen in der Sekunde eingeben kann. Für den Mikroprozessor dagegen erscheint die Zeit zwischen zwei Interrupts ewig lange, da er mit einem Takt von ca. 980 000 Schlägen pro Sekunde läuft und ein Maschinenbefehl im Durchschnitt 3 bis 4 solche Schläge zur Ausführung benötigt. Der Prozessor kann also Tausende von Instruktionen durchführen, ehe ein Interrupt ihn

aus seiner Arbeit reißt. Nach der Tastaturabfrage macht der Prozessor an der Stelle weiter, an der er vor dem Interrupt aufgehört hat.

Leider hat die Interruptroutine eine unangenehme Nebenwirkung. Sie verändert bei jedem Durchlauf bestimmte Bytes im Speicher, die für unsere Zwecke vielleicht anders aussehen sollten. Daher ist es auch vom BASIC aus nicht ohne weiteres möglich, auf das RAM in den Adressbereichen zuzugreifen, die vom ROM überlagert sind - doch davon später mehr.

Machen wir zum Schluß noch einen kleinen Test. Eine FOR-NEXT-Schleife, wie sie unten aufgelistet ist, benötigt ca. 46 Sekunden Laufzeit. Schalten wir den Interrupt jedoch ab (wie das genau funktioniert, behandeln wir in einem späteren Kapitel), so läuft das Ganze immerhin eine Sekunde schneller! Die höhere Geschwindigkeit können Sie allerdings nicht mit dem TI$ des CBM-BASIC messen, da die Interruptroutine auch für das Weitersetzen der Uhr zuständig ist.

Bevor Sie das kleine Programm (siehe unten) eintippen, sollten Sie den POKE-Befehl aus Zeile 10 im Direktmodus ausprobieren. Wenn der Cursor verschwindet und die Tastatur nicht mehr abgefragt wird, ist der Interrupt ausgeschaltet - jetzt rettet Sie nur noch RUN/STOP-RESTORE.

```
10 POKE 56334, PEEK (56334) AND 254: REM Interrupt aus
20 FOR I= 1 TO 1000: PRINT I: NEXT I
30 POKE 56334, PEEK (56334) OR 1: REM Interrupt ein
```

1.3. Wie arbeitet der Interpreter?

Wie schon gesagt, ist der BASIC-Interpreter für die Abarbeitung der BASIC-Befehle zuständig. Für den Benutzer, der davon nur das Ergebnis (nämlich den Programmablauf) sieht, ist es interessant zu erfahren, wie das funktioniert.

Beginnen wir bei der Eingabe der Befehle. Der 64er speichert unsere BASIC-Zeilen nicht einfach als Folge von Buchstaben. Das würde viel zuviel Speicherplatz beanspruchen, für den PRINT-Befehl alleine 5 Bytes (eines für jeden Buchstaben).

Vielmehr werden alle Befehlswörter als sogenannte TOKENS gespeichert, d.h. sie werden in einen Code ähnlich dem ASCII übersetzt. Zahlen und Buchstaben, die keinen Befehl bilden, werden dagegen direkt im ASCII-Code abgespeichert. Daher belegt der Befehl PRINT I auch nur zwei Bytes, eines für den Befehl, das andere für den Variablennamen. Damit ist der erste Teil der Übersetzung auch schon getan, und, so unglaublich es klingen mag, dies alles und noch mehr geschieht in der "Zeitspanne" zwischen dem Druck auf die RETURN-Taste am Zeilenende und dem Wiedererscheinen des Cursors. Oft muß dabei auch noch der gesamte Programmtext im Speicher verschoben werden (wenn eine neue Zeile nachträglich eingefügt wird).

Der zweite Teil der Übersetzung läuft ab, nachdem wir RUN eingegeben haben. Anhand der TOKENS springt der Interpreter in seine verschiedenen Unterroutinen, die dann die eigentliche Arbeit übernehmen. Beim PRINT-Befehl wären dies z.B. die Unterprogramme für "Ausdruck auswerten" (die Variable, die auf dem Bildschirm erscheinen soll) und "Zeichen auf den Bildschirm bringen", das für jedes auszugebende Zeichen einmal angesprungen wird. Für andere Befehle gibt es weitere Unterprogramme im ROM, so z.B. Arithmetik-Routinen und ähnliches.

Natürlich könnte man sich diese Routinen jetzt näher ansehen und sie analysieren, doch dies würde den Rahmen dieses Buches sprengen. Wer jedoch Interesse daran hat, sollte sich mit dem DATA-BECKER-Buch "64-intern" befassen. Es umfaßt unter anderem ein komplettes ROM-Listing des Interpreters und des Betriebssystems und enthält viele nützliche Maschinensprache-programme.

1.4. PEEK, POKE und andere Gemeinheiten

Stellen Sie sich folgende Situation vor: Sie finden in einer der zahlreichen Computerzeitschriften ein Superprogramm zum Eintippen. Sie haben inzwischen das 20 K Listing eingetippt, doch der Probelauf endete vorzeitig mit einem ERROR.

Es hilft nichts, Sie müssen die Funktionsweise des Programms verstehen, um den Fehler zu beseitigen, wenn Sie nicht jeden Buchstaben im Listing einzeln vergleichen wollen. Wenn da nur nicht diese POKE-Befehle wären! Die benutzt doch kein normaler Programmierer! Es ist also an der Zeit, das Pseudo-Geheimnis um solche Instruktionen zu lüften.

1.4.1. PEEK & POKE

Nehmen wir zuerst den POKE-Befehl. Seine Syntax dürfte bekannt sein: POKE Adresse, Byte. Die Adresse darf zwischen 0 und 65535 liegen, das Byte zwischen 0 und 255. Die Aufgabe dieses Befehls ist es, das Byte unter der angegebenen Adresse abzuspeichern. Dies kann vielen Zwecken dienen, je nach Adresse kann man damit den Bildschirm füllen, eine Farbe festlegen oder anderes mehr. Z.B. schreibt POKE 1024,1 den Code für ein A in das erste Byte des Bildschirmspeichers (falls es nicht sichtbar wird, betätigen Sie bitte die HOME-Taste, dann erscheint es bei jedem Cursorblinken), während POKE 53280,1 den Rahmen schwarz einfärbt. Damit können wir dem Computer ganz schön ins Handwerk pfuschen, denn sowohl Betriebssystem als auch Interpreter müssen sich zwangsläufig irgendwo bestimmte Daten "merken". Wie diese Daten aussehen, erfahren wir durch PEEK. Auch hier dürfte die Syntax hinlänglich bekannt sein: PRINT PEEK (Adresse) gibt das unter der angegebenen Adresse abgespeicherte Byte aus.

Wichtig ist, daß PEEK eine Funktion ist und deshalb nur innerhalb einer Zuweisung (A=PEEK...) oder eines anderen Ausdrucks stehen darf.

Gemeinsam haben beide Befehle die Eigenart, daß sich der Zweck nach der Adresse richtet, auf die zugegriffen werden soll. Es empfiehlt sich daher, bei solchen Befehlen im Speicherbelegungsplan nachzusehen, in welchem Bereich gearbeitet wird. Meist läßt sich daraus die Funktion entnehmen.

1.4.2. SYS & USR

Kommen wir nun zu den Befehlen, die eigentlich nur für Maschinenprogrammierer interessant sind: SYS Adresse und PRINT USR (x). Beide dienen zum Aufruf von Programmen in Maschinensprache.

Beim SYS-Befehl gibt die Adresse das Byte an, mit dem die Ausführung des Programmes beginnen soll. Nach Beendigung der Maschinenroutine kehrt der Interpreter wie aus einem Unterprogramm in das BASIC-Programm zurück.

Die USR-Funktion läuft ähnlich ab, aber es gibt nützliche und wichtige Erweiterungen. Der erste Unterschied liegt in der Syntax. USR ist eine Funktion wie PEEK und SIN und muß daher innerhalb eines Ausdrucks stehen (aber das kennen Sie bereits).

Außerdem brauchen Sie keine Start-Adresse anzugeben. Diese wird in einem "elektronischen Briefkasten" in den Speicherzellen 785 und 786 übergeben. Immer, wenn der Interpreter eine USR-Funktion ausführen soll, sieht er in den besagten Bytes nach, wo das Maschinenprogramm steht, das er dann ausführt. Danach kehrt er wieder ins BASIC zurück.

Das wichtigste ist aber die Möglichkeit, Daten an das Maschinenprogramm zu übergeben und umgekehrt. Der Wert in den Klammern wird zu diesem Zweck vom Interpreter in den sogenannten Fließkommaakkumulator (97 - 101) gebracht. Der Fließkommaakkumulator ist ein internes Rechenregister, in dem alle arithmetischen Operationen durchgeführt werden. Von dort kann sich das selbstgeschriebene Maschinenprogramm die Zahl abholen und bearbeiten. Nach dem Ende der USR-Routine wird die Zahl, die dann im Fließkommaakkumulator steht, an das

BASIC übergeben. Auf diese Weise kann man von Maschinensprache aus Zahlen an Variablen schicken (per A= USR (x)).

Sinn dieser Einrichtung ist es, dem Maschinenprogrammierer die Definiton eigener schneller Funktionen (z.B. Fakultät oder Sortierfunktionen) zu ermöglichen. So gesehen stellt die USR-Funktion eigentlich einen Superbefehl dar.

1.4.3. Ein kleiner Ausflug in die Binärarithmetik

Zu den beschriebenen Befehlen gesellen sich noch einige, die Sie bestimmt schon kennen, aber deren Vielseitigkeit Ihnen bisher verborgen blieb. Als erstes sind hier AND, OR und NOT zu nennen. Bisher haben Sie sie immer nur in IF-THEN-Konstruktionen verwendet, z.B. in dieser Form: IF A=0 AND B=0 THEN 100

Eigentlich sind sie aber für die logische Verknüpfung von Variablen und Zahlen gedacht. Dazu muß man wissen, daß der Rechner auch Vergleiche wie Zahlen behandelt. Probieren Sie einmal folgende Befehle:

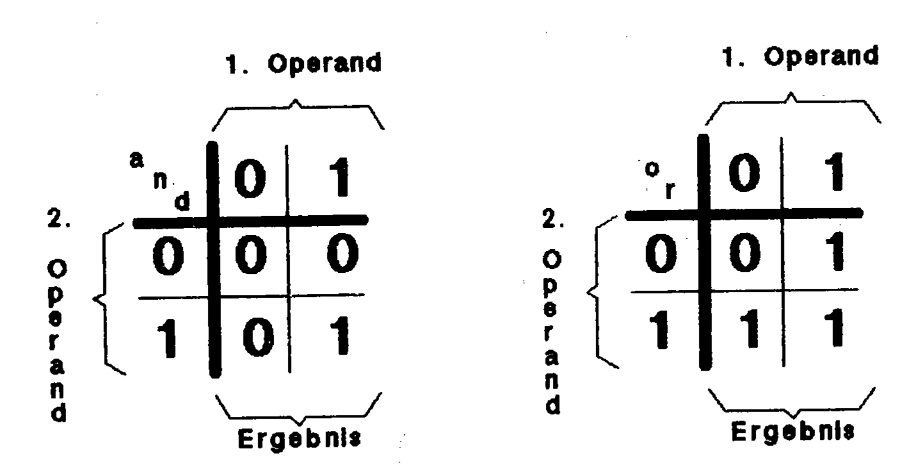
```
PRINT (1=2)
PRINT (1=1)
```

Ein Vergleich mit dem Ergebnis "wahr" liefert eine -1, ein "falsch" eine 0. Im Binärsystem sieht eine -1 so aus: 1111 1111. Bei Binärzahlen wird das am weitesten links stehende Bit meist als Vorzeichen benutzt. Ist es 1, so zeigt dies eine negative Zahl an. Tut man dies nicht, so ergibt die gleiche Kombination 255. Was aber haben die BASIC-Befehle damit zu tun?

Eine IF-THEN-Konstruktion wird immer dann verlassen, wenn das Ergebnis des Terms gleich 0 ist. Es wäre also auch folgende Befehlsfolge denkbar: IF 3*A THEN 110

Die Ergebnisse der einzelnen Vergleiche werden einfach miteinander verknüpft, und das Ergebnis daraus bestimmt den weiteren Programmablauf. Um die Wirkungsweise der einzelnen Verknüpfungen zu verstehen, machen wir jetzt einen kleinen Ausflug in die Binärarithmetik.

AND, OR und NOT sind sogenannte BOOLESCHE OPERATIONEN, die der Verknüpfung von logischen Zuständen dienen. Und wie Sie wissen, können logische Zustände mit Bits sehr einfach dargestellt werden (0 für "falsch", 1 für "wahr").
Jeweils zwei Bits werden miteinander verknüpft. Was dabei herauskommt, geben die Tabellen an.

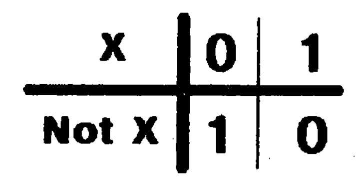


and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

Sie sehen, daß das Ergebnis auf jeden Fall dann 1 ist, wenn beide Eingangsbits 1 sind. Man kann beide Funktionen wörtlich übersetzen. Bei AND ist das Ergebnis dann 1, wenn Bit 1 und Bit 2 auf 1 sind, bei OR, wenn Bit 1 oder Bit 2 auf 1 sind.

Anders verhält es sich mit NOT. Diese Funktion kehrt einfach das Eingangsbit um.



X	0	1
Not X	1	0

So weit so gut. Leider bleibt für uns unbedarfte BASIC-Programmierer noch ein Problem. Im BASIC nützen uns einzelne Bits wenig. Dort haben wir es mit Dezimalzahlen zu tun. Um zu berechnen, was der Ausdruck 45 AND 123 ergibt, müssen wir folgendermaßen vorgehen:

1. Zahl in Dualsystem umwandeln

Das geht einfacher als Sie denken. Sie müssen die Dezimalzahl nur fortwährend durch 2 teilen und den Rest jeder Division als Bit notieren, bis das Ergebnis 0 wird.

Beispiel:

23	/	2	=	11	Rest	1
11	/	2	=	5	Rest	1
5	/	2	=	2	Rest	0
2	/	2	=	1	Rest	0
1	/	2	=	0	Rest	1

$$\rightarrow 23_{10} = 10111_2$$

Auch der umgekehrte Weg ist sehr einfach. Hier ein Beispiel:

$$10111_2 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$$

$$= 23_{10}$$

Aber zurück zum Thema. Die Zahlen 45 und 123 sehen im Binärsystem also so aus:

```
 45 = 00101101
123 = 01111011
```

2. Dualzahlen bitweise verknüpfen

In unserem Beispiel 45 AND 123 ergibt das:

```
         45  =  00101101
AND     123  =  01111011
------------------------
=        41  =  00101001
```

3. Ergebnis in Dezimalsystem umwandeln

00101001 = 41

Das könnten Sie natürlich auch einfacher haben, indem Sie einfach PRINT 45 AND 123 eintippen. Aber so hat man ungleich mehr Einblick.

Mit Recht stellen Sie jetzt die Frage, wozu das gut sein soll. Neben der Verknüpfung von Vergleichen werden diese Befehle oft zum Beeinflussen einzelner Bits benutzt. Z.B. wird durch eine AND-Verknüpfung mit 254 auf jeden Fall das am weitesten rechts stehende Bit gelöscht.

	254	=	11111110
AND	123	=	01111011
=	122	=	01111010

Weil die 7 linken Bits alle mit 1 verknüpft werden, bleiben sie erhalten; das ganz rechte Bit wird immer 0, weil es auch in 254 auf 0 ist.

Ähnlich kann man das rechte Bit mit OR 1 wieder setzen:

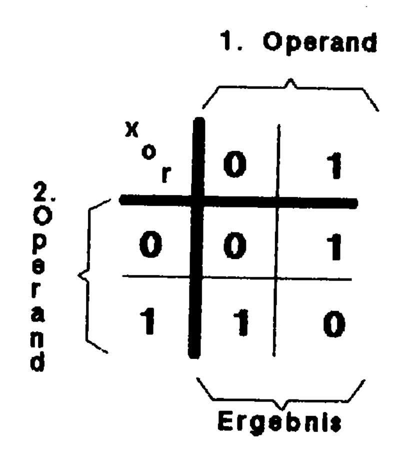
	1	=	00000001
OR	122	=	01111010
=	123	=	01111011

Wie auch immer das 8. Bit im zweiten Operand aussieht, im Ergebnis steht auf jeden Fall eine 1. Probieren Sie es mit beliebigen Zahlen aus.

Jetzt bleibt nur noch ein geheimnisvoller Befehl: WAIT Adresse, X, Y.

Er hat eine Aufgabe, bei der sich jedem Prozessor die Bits im Speicher umdrehen. Er soll nämlich warten. Und das mag ein Computer überhaupt nicht. Das Warten wird durch fortlaufende Verknüpfung von Bytes erzeugt. Kommt der Interpreter zu einem WAIT-Befehl, so liest er zunächst den Inhalt der angegebe-

nen Speicherzelle. Diese Zahl wird EXKLUSIV-ODER mit der Zahl Y verknüpft. Wie der Name schon andeutet, ist XOR (Kurzform für EXKLUSIV-ODER) ein Verwandter der ODER-Verknüpfung. Die Tabelle zeigt die Arbeitsweise.



Das Ergebnis wird nur dann 1, wenn entweder Bit 1 oder Bit 2 auf 1 sind, nicht aber beide.

Das Ergebnis der ersten Verknüpfung wird nun noch AND-verknüpft mit der Zahl X. Sollte dieses Ergebnis 0 sein, so wiederholt der Interpreter die ganze Prozedur, andernfalls macht er mit dem nächsten Befehl weiter.

Es gibt aber auch noch eine zweite Variante des WAIT-Befehls, bei der das Y-Argument nicht angegeben wird. Hier wartet der Interpreter, bis der Inhalt der angegebenen Speicherzelle AND X ungleich 0 wird.

Es leuchtet ein, daß dieser Warteprozeß nur dann abgebrochen wird, wenn die entsprechende Speicherzelle vom Interpreter oder

Betriebssystem verändert wird. Während eines WAIT-Befehls können wir ja nicht POKEn!

1.5. Der Aufbau des Rechners

Keine Angst, auch hier wird es nicht zu technisch. Es ist für das Verständnis einiger Tricks sehr nützlich, wenn man etwas über das Innenleben des 64ers weiß.

Sie werden sich sicher schon gewundert haben, daß Commodore mit der Angabe "64 k RAM" wirbt, für das BASIC aber nur 38 K zur Verfügung stehen.

Nun, die 64 Kilobytes sind tatsächlich vorhanden, doch Sie können sie nicht direkt nutzen. Der Mikroprozessor 6510 kann insgesamt nur 64 K adressieren, d.h. er kann 65536 verschiedene Speicherplätze ansprechen. Mit dem RAM wären also die Möglichkeiten des Rechners voll ausgenutzt. Doch leider braucht ein Computer auch ROMs und Speicherplätze für interne Funktionen. Zum einen ist da die schon erwähnte Zeropage, zum anderen die ROMs mit dem BASIC-Interpreter, dem Betriebssystem und dem Charaktergenerator (was das ist, und wie er funktioniert, sehen wir später; vorläufig reicht die Information, daß hier die Formen der Bildschirmzeichen gespeichert sind).

Der ROM-Bereich belegt 20 K. Von unseren 64 K bleiben also noch ganze 44, von denen wir noch zwei Kilobytes für Videoram und Zeropage sowie 4 K freies RAM ab Speicherzelle 49152 abziehen. Es bleiben die erwähnten 38 Kilobytes.

Abb. 2. zeigt ein vereinfachtes Blockschaltbild des Rechners. Wie Sie sehen, liegen ROM und RAM übereinander und belegen den gleichen Adressbereich. Um zwischen beiden umschalten zu können, muß man die Speicherzelle 1 verändern. Hier wird festgelegt, ob ROM oder RAM benutzt werden können. Leider ist dies vom BASIC aus nicht so ohne weiteres möglich. Würden wir nämlich das BASIC-ROM abschalten, so fände der Prozessor sein Programm nicht mehr vor, sondern nur noch den leeren Speicher. Die Folge davon wäre ein Aufhängen des Rechners

und ein recht überzeugender Aufschrei des Benutzers ob der verlorenen Daten.

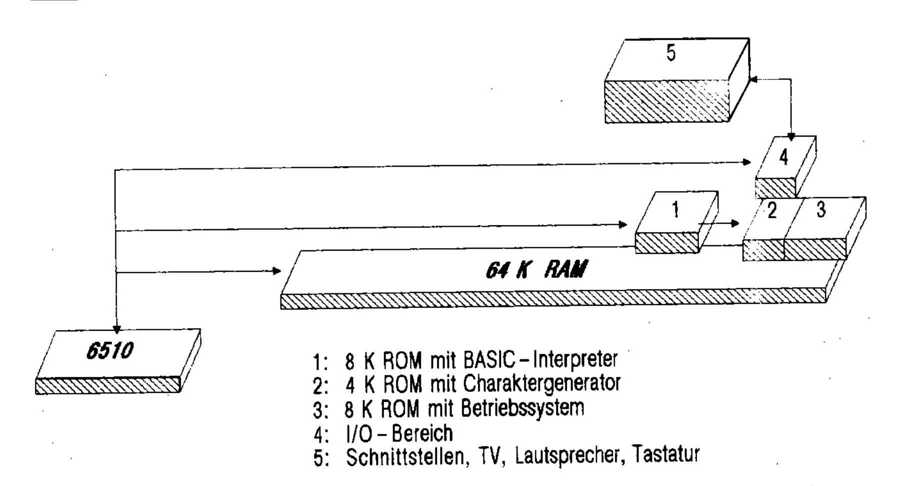
Eine kleine Hilfe gibt uns der CBM trotzdem. Vom BASIC aus ist nämlich das Schreiben in die überlagerten Bereiche mittels POKE oder LOAD jederzeit möglich, lediglich das Lesen durch PEEK oder SAVE kann nicht durchgeführt werden. Das bedeutet, daß ein POKE, der eine Speicherzelle im ROM adressiert, das darunterliegende RAM beeinflußt, ein PEEK mit der gleichen Adresse dagegen tatsächlich das ROM ausliest.

Es bleiben noch die 4 K RAM im oberen Drittel des Adressbereiches. Sie sind für die Programmierung in Maschinensprache gedacht, können aber auch von BASIC-Programmen mittels PEEK und POKE als Datenspeicher genutzt werden.

Schließlich gibt es noch den sogenannten I/O-Bereich. I steht für INPUT, O für OUTPUT. Hier liegen nämlich die Bausteine, die für die Schnittstellen, die Tastatur, den Bildschirm und die Tonerzeugung zuständig sind. Der Prozessor liefert hier seine Daten ab, die der entsprechende Baustein dann beispielsweise zu einem TV-Signal, einem Ton oder einem Floppyzugriff verarbeitet. Umgekehrt kommen hier auch die Daten von der Tastatur oder den Peripheriegeräten an. Außerdem befindet sich hier noch das COLOR-RAM.

Wie das Blockbild zeigt, "stapeln" sich hier die Bytes sogar dreifach, da die I/O-Bausteine jeweils eigene "RAM-Abteilungen" besitzen. Wenn wir also auf die Speicherzelle 53280 zugreifen, um die Rahmenfarbe zu ändern, dann wird das Byte nicht in das RAM, sondern in das Register des Bausteines selbst geschrieben. Dies gilt auch für das COLOR-RAM.

Der 64er besitzt insgesamt 4 I/O-Bausteine. Zwei davon kennen Sie, nämlich den VIC (der für die Grafik und den Bildschirm zuständig ist) und den SID (der für die Tonerzeugung eingesetzt wird). Es bleiben noch zwei CIAs (Complex Interface Adapter), die die Tastatur und einige Schnittstellen wie z.B. den USER-PORT bedienen.



1.6. Für eigene Experimente: Resettaster

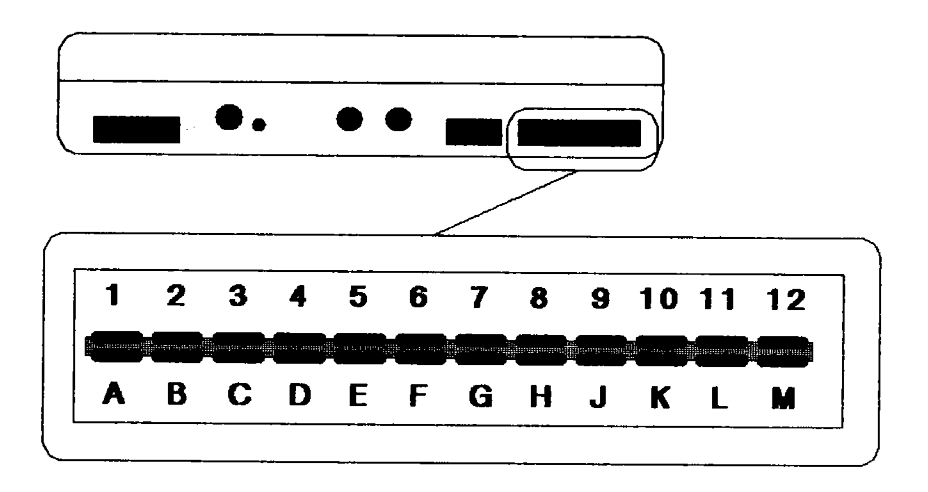
Wenn Sie selbst mit PEEK und POKE experimentieren wollen, kann es passieren, daß sich der Rechner "aufhängt". In vielen Fällen läßt sich dies mit der Tastenkombination RUN/STOP-RESTORE beheben. Oft ist dies jedoch nicht möglich, so daß das Ausschalten des Rechners unumgänglich erscheint, will man den 64er aus seiner "stabilen Umlaufbahn um den Saturn" herausholen. Dabei gehen eventuell benutzte Hilfsprogramme (wie z.B. Hex-Monitor o.ä.) verloren. Um dies zu vermeiden, ist der Selbstbau eines RESETTASTERS anzuraten. Man benötigt hierzu einen USER-PORT-Stecker und einen einfachen Taster. Der Stecker ist leider nicht gerade ein Pfennigartikel. Trotzdem sollte man vor dieser Ausgabe nicht zurückschrecken, da sich der USER-PORT sehr vielseitig einsetzen läßt und der Stecker mehrfach genutzt werden kann.

Der Taster wird mit den Pins 1 und 3 (siehe Abb. 3.) des USER-PORTs verbunden. Wird der Stromkreis geschlossen, so führt der Prozessor einen RESET aus, das heißt, er bringt den Rechner in den Einschaltzustand. Dabei werden jedoch nur vom Betriebssystem benötigte Bytes verändert. Vom BASIC aus kann dies auch per Software durch SYS 64738 ausgelöst werden.

Eventuell im Speicher befindliche Maschinenspracheprogramme wie z.B. SIMONS BASIC oder Assembler bleiben erhalten, da die Spannung ja nicht abgeschaltet wird, und können ggf. mit SYS xxxxx (vorher Startadresse merken!) wieder gestartet werden. Besitzt man ein Programm, mit dem der NEW-Befehl rückgängig gemacht werden kann, so lassen sich sogar BASIC-Programme wieder restaurieren.

Aber Vorsicht! Einer meiner Bekannten hat einmal mangels USER-PORT-Stecker eine Büroklammer zum Auslösen des RESETs benutzt. Sein Pech: Er hat statt der Pins 1 und 3 die Pins L und N erwischt. Er flucht noch heute...

Eine Warnung noch an Besitzer von Diskettenlaufwerken. Wird ein RESET des Rechners ausgelöst, so wird auch die Floppy neu initialisiert. Daher sollte man vorher eine evtl. noch im Gerät steckende Diskette herausnehmen, um Unheil zu vermeiden.



2. Die Zeropage

2.1. Die Zeropage ist keine Null

Wenn Sie sich schon einmal den Anhang des CBM-Handbuches angesehen haben, so ist Ihnen sicher der Abschnitt mit der Zeropage aufgefallen. Sie bietet dem Anwender eine wahre Fundgrube mit Tricks und neuen Programmiermöglichkeiten - man muß sie nur zu nutzen wissen.

Der Name Zeropage ist übrigens nicht ganz richtig. Üblicherweise bezeichnet man damit die ersten 256 Bytes des Adressbereiches eines Mikroprozessors. Das kommt daher, daß man den gesamten Adressbereich des 6510 in sogenannte Speicherseiten (engl. RAM-Pages) zu 256 Bytes einteilen kann. Wie Sie wissen, kann ein Byte 256 Speicherzellen adressieren. Da aber eine Adresse normalerweise aus zwei Bytes besteht, kann man sagen, daß das erste Byte die Seite numeriert. Die Zeropage liegt ganz am Anfang des Speichers, also ist es die nullte Seite (= 0-Page). Bei Commodore-Rechnern ist jedoch meistens das erste Kilobyte gemeint.

Sinn und Zweck des Ganzen ist schnell erklärt. Betriebssystem und Interpreter brauchen Register, um sich Zustände, Zahlen oder Codes merken zu können. Wie Schulkinder bei der Addition "1 im Sinn" behalten, so tut dies der Computer in der sog. Zeropage. Die ersten 256 Bytes sind gut für die schnelle Speicherung von Daten geeignet, da sie mit nur einem einzigen Byte (und daher besonders schnell) adressiert werden können.
Viele Register in der Zeropage müssen eine bestimmte Zahl enthalten, um ein ordnungsgemäßes Funktionieren des Rechners zu gewährleisten. Andere werden gar nicht benutzt und stehen uns zur freien Verfügung. Weitere Bytes können vom Anwender sinnvoll und wirksam beeinflußt werden.

2.2. Pointer & Stacks

Zwei Fachbegriffe, auf die Sie immer wieder stoßen werden, sind Pointer und Stack.

Pointer (engl. Zeiger) zeigen auf bestimmte Stellen im Speicher und werden auch Vektoren genannt (welchen der Begriffe Sie benutzen, ist völlig egal). Dort können entweder Informationen oder Unterprogramme stehen. Der Cursorpointer beispielsweise zeigt auf das Byte im Bildschirmspeicher, die die Stelle repräsentiert, wo der Cursor blinkt. Will man das Zeichen unter dem Cursor ändern, so holt man sich die Adresse des dafür zu ändernden Bytes einfach aus dem Zeiger und schreibt ein anderes Zeichen in den Bildschirmspeicher.

Zeiger auf Unterprogramme wurden eingeführt, um Erweiterungen des Interpreters zu ermöglichen. Würden alle Unterprogramme direkt durch ROM-Befehle angesprungen, so könnte das BASIC nicht erweitert werden. Hier aber holt sich der Mikroprozessor die Adresse eines Unterprogramms aus dem RAM, d.h. der POINTER kann verändert werden. So können wir ihn auf ein eigenes Unterprogramm zeigen lassen. Ändern wir den Vektor auf die Routine für Zeichenausgabe, so ist es z.B. möglich, mittels einer eigenen Maschinenroutine den PRINT-Befehl so zu ändern, daß jedes Zeichen gleichzeitig auf Bildschirm und Drucker erscheint.

Pointer haben immer ein bestimmtes Format. Sie bestehen im allgemeinen aus zwei Bytes, wovon das erste LOWBYTE (niederwertiges Byte) und das zweite HIGHBYTE (höherwertiges Byte) genannt werden. Um die Position des Cursors oder das Byte zu erhalten, auf das gezeigt wird, benutzt man folgende Formel:

Adresse = Lowbyte + 256 * Highbyte

Es gehört zu den Besonderheiten der Computer, daß das Lowbyte immer vor dem Highbyte im Speicher steht.

Eine Besonderheit stellen die Ein-Byte-Pointer dar. Sie geben nicht die konkrete Adresse eines Unterprogramms oder Wertes an, sondern zeigen innerhalb eines bestimmten Bereichs (z.B. Tastaturpuffer, Stack) auf die aktuelle Position (0 - 255) und werden zu einer BASISADRESSE addiert.So beginnt z.B. der Stapel (siehe unten) bei Byte 256 und endet bei 511. Der Stackpointer (ein Byte) zeigt dann z.B. auf das 12. Element. Dessen Adresse erhält man durch

$$256 + 12 = 268$$

Ein Stack (engl. Stapel) hat die Aufgabe, Daten zwischenzuspeichern und in der umgekehrten Reihenfolge wieder zurückzugeben, wenn sie benötigt werden. Wie bei einem richtigen Stapel kann man immer nur das oberste Element herunternehmen und auch nur ganz oben neue Daten dazulegen. Dies wird vor allem für Unterprogramme benötigt. Beim Aufruf des Unterprogramms wird die gegenwärtige Stelle im Programm auf dem Stack zwischengespeichert, beim RETURN holt sich der Rechner diese Adresse zurück und setzt das Programm fort (siehe Abb. 4). Übrigens ist der Stapel dafür verantwortlich, daß Sie höchstens eine bestimmte Anzahl von Unterprogrammen verschachteln können. Irgendwann ist nämlich der Stapel voll und kann sich keine weiteren Rücksprungadressen mehr merken.

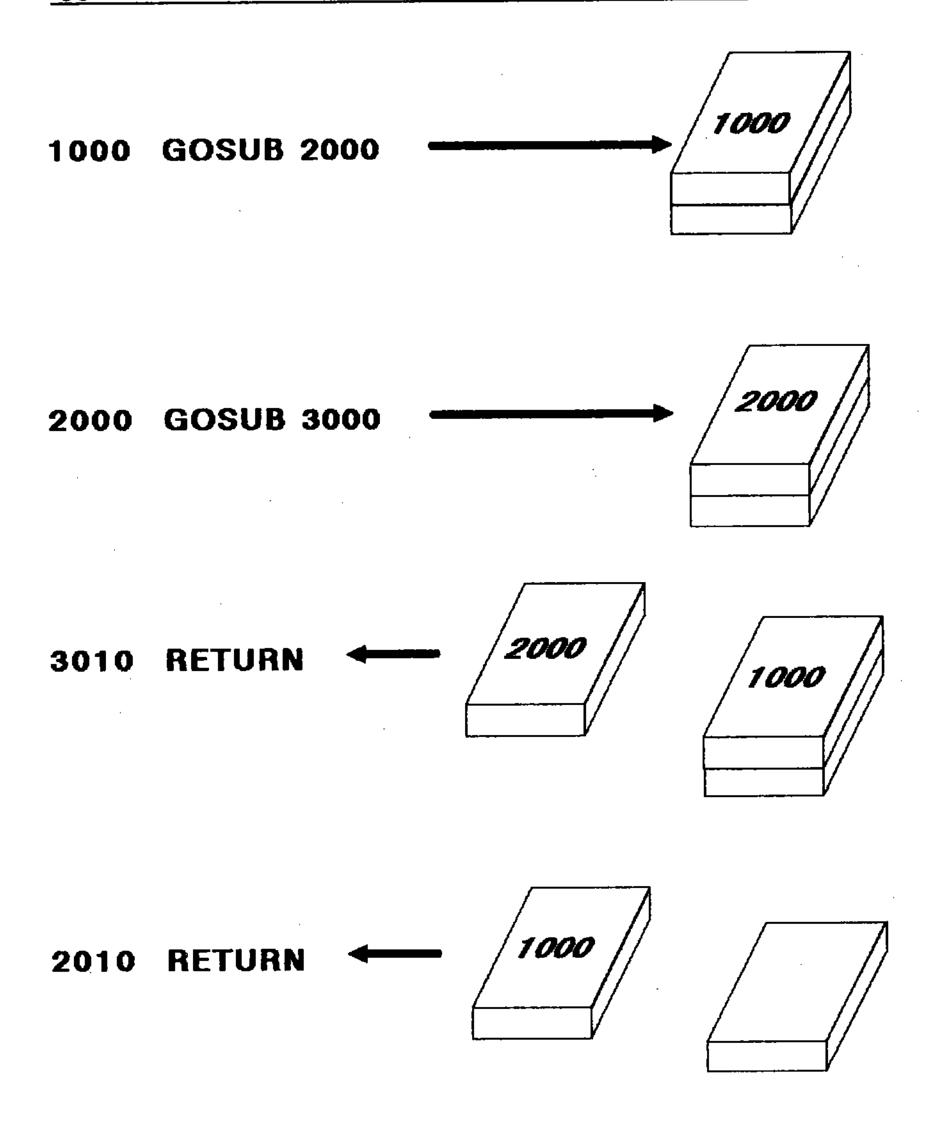


Abb. 4: Stapelzugriff

Der 64er hat neben anderen Interpreterstacks für Variablen u.ä. drei solche Stapel, die nicht verändert werden sollten.

1. Prozessorstack (256 - 511) für Maschinensprache
2. Stack für BASIC-Unterprogramme
3. Stack für FOR-NEXT-Schleifen

Wundern Sie sich nicht, wenn Sie die beiden Letztgenannten im Anhang des Handbuches und im Speicherbelegungsplan nicht finden werden. Sie sind in "verschieden genutzten Bereichen" versteckt. Das sollte uns aber nicht weiter stören denn ein Verändern dieser Stacks hat außer einem kapitalen Absturz keine weiteren Folgen.

Nach soviel Theorie werden wir nun in die Praxis einsteigen. In den nächsten Abschnitten finden Sie (neben nötigen theoretischen Abhandlungen) interessante Tricks, die Ihnen bei der Programmierung Ihres 64ers helfen können.

Zusammenfassung: Zeiger

Adresse = Lowbyte + 256 * Highbyte
Lowbyte = Adresse - INT(Adresse/256)*256
Highbyte = INT(Adresse / 256)
Zeiger bestehen im Normalfall aus zwei Bytes, die immer in der Reihenfolge LOW /HIGH angeordnet sind.

3. Der Speicher

3.1. Der Speicherbelegungsplan

Im Kapitel 14 finden Sie einen genauen Speicherbelegungsplan des CBM-64. Neben der Zeropage ist auch der I/O-Bereich aufgelistet. Besonderes Augenmerk sollten Sie Abweichungen von der Zeropageliste im CBM-Handbuch widmen. So sind z.B. die ersten 5 Bytes des angeblich freien Bereichs von 673 bis 767 durch die CIAs belegt. Es ist also Vorsicht geboten, wenn man die Zeropage als Datenspeicher benutzen will. Ein falscher POKE kann zum Abstürzen des Interpreters führen.

Trotzdem sollten Sie sich dadurch nicht vom Experimentieren abhalten lassen. Viele Tricks wurden durch Zufall entdeckt, andere traten nach gezielter Suche zutage. Soweit ich weiß, kann keine POKE-Kombination den Rechner zu Schäden führen. Viel Spaß also beim Experimentieren!

3.2. Das magische Byte 1

Magisch ist das Byte, weil es - wie schon angesprochen - die Speicheraufteilung steuert. Dabei werden allerdings nur die Bits 0 - 2 eingesetzt. Im Normalfall sind alle drei Bits auf 1. Wird eines dieser Bits auf 0 gesetzt, so ändert sich die Speicheraufteilung entsprechend.

Mit Bit 0 kann das BASIC-ROM (40960 - 49151) abgeschaltet werden, mit Bit 1 werden BASIC und Betriebssystem gleichzeitig abgeschaltet. Sind beide Bits auf 0, so wird außerdem noch der I/O-Bereich abgeschaltet, d.h. es stehen jetzt 62 K zur Verfügung (da Zeropage und TV-RAM nicht überlagert werden). Bit 2 bestimmt schließlich, ob der Charaktergenerator ausgelesen werden kann (Sie erinnern sich: dreifache Belegung).

Leider hat dieses System einen Haken. Schalten wir BASIC und Betriebssystem ab, so hängt sich der Rechner auf. Daher können

wir nur über die Maschinensprache auf das recht überzeugend versteckte RAM zugreifen.

Etwas anders verhält es sich beim Charaktergenerator. Hier befindet sich kein Programm, das zum Betrieb des Rechners notwendig wäre. Trotzdem hängt sich der 64er auf, wenn Bit 2 auf 0 gesetzt wird, da damit automatisch nicht mehr auf den I/O-Bereich zugegriffen werden kann. Nichts anderes aber tut die Interruptroutine, um die Tastatur abzufragen. Hier hilft es, wenn wir den Interrupt nach bekanntem Muster (siehe Kap. 1.2.) ausschalten.

Um Ihnen die Benutzung von stolzen 62 Kilobytes wenigstens mittels PEEK und POKE zu ermöglichen, muß ich mein im Vorwort gegebenes Versprechen brechen und Ihnen ein winziges Maschinenspracheprogramm vorstellen, dessen Entsprechung in BASIC zwar programmiert werden kann, aber nicht funktioniert. Ich habe das BASIC-Programm der leichteren Verständlichkeit wegen trotzdem aufgelistet:

```
1 REM Achtung! Auf keinen Fall starten!
10 POKE 56334, PEEK (56334) AND 254: REM Interrupt aus
20 POKE 1, PEEK (1) AND 252: REM ROM abschalten
30 POKE 2, PEEK (PEEK (251) + 256 * PEEK (252))
40 POKE 1, PEEK (1) OR 3: REM ROM einschalten
50 POKE 56334, PEEK (56334) OR 1: REM Interrupt an

30a POKE (PEEK (251) + 256 * PEEK (252)), PEEK (2)
```

Und hier das Maschienenprogramm:

```
10 DATA 120, 165, 1, 41, 252, 133, 1, 160, 0, 177, 251, 133, 2,
165, 1, 9, 3, 133, 1, 88, 96
20 DATA 120, 165, 1, 41, 252, 133, 1, 160, 0, 165, 2, 145, 251,
165, 1, 9, 3, 133, 1, 88, 96
30 FOR I= 680 TO 721: READ A: POKE I, A: NEXT I
```

Sehen wir uns die Programme näher an. Die Zeilen 10 und 50 müßten Ihnen bekannt vorkommen. In Zeile 20 werden die Bits 0 und 1 von Register 1 gelöscht und damit 62 K RAM nutzbar gemacht. In Zeile 30 wird das gewünschte Byte ausgelesen und in Byte 2 zwischengespeichert, damit wir es als Benutzer später abholen (sprich PEEKen) können. Die Adresse des anzusprechenden Bytes ist als Zeiger in den Speicherstellen 251 und 252 angelegt. Der Term innerhalb der Klammern des ersten PEEK-Befehls berechnet aus LOW- und HIGHBYTE die Adresse.

Nehmen wir an, Sie wollen das Byte 56000 (übrigens liegt es unter dem COLOR-RAM) ansprechen. Das Highbyte des Zeigers berechnet sich folgendermaßen:

```
HIGHBYTE = INT (56000 / 256)
```

Um das Lowbyte zu erhalten, blenden wir einfach das höherwertige Byte aus der 16-Bit-Zahl 56000 aus:

```
LOWBYTE = 56000 - INT (56000 / 256)*256
```

Um den Zeiger zu setzen, benutzen wir folgende Befehle:

```
POKE 251, LOWBYTE: POKE 252, HIGHBYTE
```

Nach SYS 680 können wir das gewünschte Byte mittels PRINT PEEK (2) ausgeben.

Ein kompletter Funktionsaufruf sieht dann so aus:

```
POKE 251, LOWBYTE: POKE 252, HIGHBYTE: SYS 680
PRINT PEEK(2)
```

Nun zu Zeile 30a. Wenn wir in das RAM unter dem I/O-Bereich schreiben wollen, so geht dies im Gegensatz zu den anderen überlagerten Bereichen nicht mit dem normalen POKE-Befehl. Also müssen wir auch hier das ROM abschalten.

Das Programm dazu ist fast identisch mit dem PEEK-Programm. Lediglich Zeile 30 wird (im BASIC) durch 30a ersetzt; das

Funktionsprinzip ist fast das gleiche. Nur muß der POKE-Wert jetzt vorher in Byte 2 abgespeichert werden. Der Zeiger auf das gewünschte Byte wird wie üblich gesetzt. Gestartet wird diesmal mit SYS 701.

Hier ein Beispielaufruf:

```
POKE 251, LOWBYTE: POKE 252, HIGHBYTE
POKE 2, POKE-WERT: SYS 701
```

Unter dem BASIC-Programm finden Sie das Ladeprogramm für die beiden Maschinenroutinen. Zeile 10 umfaßt das gesamte PEEK-Programm, Zeile 20 das POKE-Programm. Die beiden Zeilen unterscheiden sich nur in 4 Bytes.

Die Routinen können unabhängig voneinander benutzt und verschoben werden, d.h. sie können dorthin gePOKEd werden (siehe Zeile 30), wo Sie es möchten. Man nennt diese Eigenschaft Relokatibilität.

Die Länge jeder Routine beträgt 21 Bytes.

Damit sind wir in der Lage, 62 K RAM zu benutzen, davon 38 K für BASIC-Programme und Variablen. Die verbleibenden 24 Kilobytes können mittels POKE beschrieben werden (Ausnahme: 4 K I/O-Bereich von 53248 bis 57343) und durch die beschriebene Routine wieder ausgelesen werden.

Weil die Handhabung des beschriebenen Programms nicht gerade komfortabel ist, folgt unten noch eine andere Version, die mit PRINT USR (Adresse) aufgerufen werden kann. Für das POKEn unter den I/O-Bereich kann man folgende Kombination benutzen:

```
SYS 715, Adresse, Byte
```

Wer sich jetzt wundert, daß eine solche ungewöhnliche Syntax zulässig ist, dem sei verraten, daß der 64er für Maschinenprogrammierer geradezu paradiesische Möglichkeiten bietet. Unter

Ausnutzung von ROM-Routinen kann man sich eigene Befehle der beschriebenen Art programmieren.

Auch dieses Programm ist relokatibel, man muß dann jedoch den USR-Vektor, der dem Interpreter mitteilt, wo die USR-Funktion steht, in Zeile 70 wieder richtig initialisieren. Dies funktioniert genau wie die Zeigerberechnung für die erste Version des PEEK-Programms. Der Zeiger muß immer auf die Adresse weisen, mit der die FOR-NEXT-Schleife aus Zeile 60 beginnt.

```
10 DATA 165, 20, 72, 165, 21, 72, 32, 247, 183, 120, 165, 1, 41,
252, 133
20 DATA 1, 160, 0, 177, 20, 168, 165, 1, 9, 3, 133, 1, 88, 104,
133, 21
30 DATA 104, 133, 20, 76, 162, 179, 32, 253, 174, 32, 138, 173,
32, 247
40 DATA 183, 32, 253, 174, 32, 158, 183, 165, 1, 41, 252, 133, 1,
138
50 DATA 160, 0, 145, 20, 165, 1, 9, 3, 133, 1, 96
60 FOR I= 678 TO 747: READ A: POKE I,A: NEXT I
70 POKE 785, 166: POKE 786, 2
```

Zusammenfassung: Speicherüberlagerung

Wird über Speicherzelle 1 gesteuert. Bits 0 - 2 im Normalfall auf 1. Beim Löschen der Bits Veränderung der Aufteilung. Bit 0 schaltet BASIC-ROM ab, Bit 1 BASIC und Betriebssystem gleichzeitig, beide Bits zusammen zusätzlich auch I/O-Bereich. Bit 2 ermöglicht Auslesen des Charaktergenerators (vom BASIC aus nach Abschalten des Interrupts möglich).

3.3. Speicher schützen

Nachdem wir uns die unendlichen Weiten eines 62-K-Speichers erschlossen haben, tun wir jetzt genau das Gegenteil: Wir verkleinern den BASIC-Speicher, um bestimmte Daten vor dem In-

terpreter zu schützen. Hier stellt sich sofort die Frage, wozu das gut sein soll. Nehmen wir an, Sie wollten ein Programm schreiben, das mit 8 verschiedenen Sprites arbeitet.

Vier davon können wir in den Blöcken 11, 13, 14 und 15 unterbringen. Aber nach Block 15 schließen sich TV-RAM und BASIC-Speicher an - beides Bereiche, die man tunlichst nicht überschreiben sollte. Konsequenz: Wir müssen den Beginn des BASIC-Programms verlegen, um Platz zu schaffen. Dazu stellt uns die Zeropage Zeiger zur Verfügung, die den Beginn bzw. das Ende des Speichers anzeigen.

Um das BASIC-Programm bei Speicherstelle 2560 beginnen zu lassen, müssen wir den Zeiger in den Speicherzellen 43/44 in der bekannten Weise ändern:

```
POKE 43, (2560 + 1) - INT(2561/256)*256
POKE 44, (2560 + 1) / 256
```

Natürlich können Sie statt der Formeln auch die Zahlen direkt einsetzen.

Die Addition von 1 ist nötig, da der Zeiger auf den Beginn der ersten Zeile weisen soll. Das erste Byte im BASIC-Programm muß 0 sein, also:

```
POKE 2560, 0
```

Es bleibt nur noch NEW übrig, um die Zeiger für Variablen, Arrays u.ä. der neuen Situation anzupassen. Diese zeigten noch auf den alten BASIC-Start bei 2048. Ein CLR reicht dafür nicht aus!

Um das Ende des BASIC-RAMs nach unten zu verlegen, gehen wir ähnlich vor. Wir benutzen allerdings die Register 55/56 und können uns das POKEn einer 0 sparen.

Leider bringt diese Methode noch einen großen Nachteil mit sich. Durch das Setzen der Zeiger auf einen neuen Bereich wird nicht automatisch auch der Programmtext im Speicher verscho-

ben. Also müssen diese Befehle vor der Eingabe bzw. dem Laden des Programms gegeben werden. Der einfachste Weg ist ein kleines Ladeprogramm, mit dem die Zeiger neu gesetzt und das Hauptprogramm geladen wird. Theoretisch könnte das so aussehen:

```
10 POKE 43, (2560 + 1) - INT(2561/256)*256
20 POKE 44, (2560 + 1) / 256: POKE 2560, 0: NEW
30 LOAD "Hauptprogramm"
```

Leider funktioniert das nicht, denn spätestens nach NEW ist Schluß. Ohne diesen Befehl werden aber die übrigen Pointer nicht neu eingestellt. Es bleibt uns nichts anderes, als alle Zeiger (auch die für die Variablen und Arrays, siehe Abb. 5) von Hand zu setzen. Dazu gehen wir wie folgt vor:

1. Speicher im Direktmodus (wie oben) schützen
2. Programm laden
3. Programm starten und darauf achten, daß alle Variablen mindestens einmal benutzt werden.
4. Durch untenstehende Befehle alle Pointer auslesen; die Zahlen merken!

   PRINT PEEK(43), PEEK(44)
   PRINT PEEK(45), PEEK(46)
   usw. bis 50.

5. Setzen Sie jetzt die Zahlen in folgendes Programm ein:

```
1 POKE 43, Zahl 1: POKE 44, Zahl 2: POKE Adresse,0
2 POKE 45, Zahl 3: POKE 46, Zahl 4: POKE 47, Zahl 5:
POKE 48, Zahl 6: POKE 49, Zahl 7: POKE 50, Zahl 8
3 CLR: LOAD "programm"
```

Damit haben Sie ein lauffähiges Ladeprogramm.

Wenn dieses Programm abläuft, dann macht der 64er eigentlich etwas Unmögliches. In den ersten beiden Zeilen werden die Zeiger für den BASIC-Speicher hochgesetzt. Damit läßt sich das Programm nicht mehr listen, eigentlich existiert es für den Interpreter gar nicht mehr. Trotzdem führt er die restlichen Zeilen noch aus. Nur Sprung- oder ähnliche Befehle kann er nicht mehr ausführen, da er in diesem Fall ab der augenblicklichen Zeigerposition nach der betreffenden Zeilennummer suchen würde.

Auch der LOAD-Befehl beinhaltet einen kleinen Trick. Wird LOAD während des Programmlaufs ausgeführt, so macht der Rechner nach dem Ladevorgang nicht im alten Programm weiter, sondern beginnt am neuen Programmanfang mit der Ausführung. Damit wird das Hauptprogramm also automatisch gestartet.

Bei der Programmerstellung sollten die Zeiger vorher von Hand hochgesetzt werden, damit die Sprites (oder anderes) nicht die mühsam eingegebenen Programmzeilen überschreiben. Sollte das Programm schon fertig sein und nur noch auf das Verschieben warten, so sollte man es auf Diskette oder Cassette abspeichern und dann mit dem beschriebenen Lader wieder in den Speicher bringen.

Wir werden in den folgenden Abschnitten noch einige Anwendungen kennenlernen, für die wir Teile des BASIC-Speichers schützen müssen.

Zusammenfassung: Speicher schützen

BASIC-Anfang hochsetzen:
POKE 43, LOW: POKE 44, HIGH: POKE Adresse, 0: NEW

BASIC-Ende heruntersetzen:
POKE 55, LOW: POKE 56, HIGH: NEW

3.4. Freier Speicher

Es ist zwar schon oft behandelt worden, dennoch sei es hier für die "Nachzügler" noch einmal erwähnt: Das Problem mit der FRE (0)-Funktion.

Ist der freie Speicher kleiner als 32768 Bytes, so erhalten wir nach PRINT FRE(0) die positive Anzahl freier Bytes. Ist der freie Bereich jedoch größer (z.B. nach dem Einschalten), so erhalten wir eine negative Zahl, die zudem nichts über die Speichergröße auszusagen scheint. Woran liegt das?

Die FRE-Funktion liefert einen Integerwert. Integer-Variablen des BASICs haben jedoch einen Wertebereich von -32767 bis +32767. Der Interpreter muß bei einer größeren Zahl (z.B. 38000) auf die negativen Werte ausweichen. Die wirkliche Anzahl freier Bytes erfahren wir durch PRINT 65538 + FRE (0), sofern FRE(0) kleiner 0 ist.

Nun zu einem anderen Thema. Oft möchte man ein paar Daten abspeichern, um sie einem Maschinenprogramm zu übergeben, oder man möchte nicht unbedingt eine ganze Variable für ein Byte oder gar ein Bit "verschwenden". Ebenfalls ist es denkbar, daß zwar in einem Programm alle Variablen gelöscht werden sollen (per CLR), eine oder mehrere Steuervariablen jedoch unbedingt erhalten werden müssen. Was tun?

Es empfiehlt sich, einen freien Bereich in der Zeropage zu suchen, um die Daten dorthin zu POKEn. Von einem CLR oder NEW werden sie dann nicht berührt. Unten finden Sie eine Liste von freien Bereichen in der Zeropage sowie Bemerkungen dazu (soweit erforderlich).

Freie Bytes

251	-	254	können evtl. verändert werden
678	-	767	
780	-	783	nur wenn kein SYS gegeben wird
820	-	827	
828	-	1019	wird bei Kassettenbetrieb überschrieben
1020	-	1023	
2024	-	2039	
49152	-	53247	

4. Massenspeicherung und Peripherie

4.1. Abspeichern von Grafiken, Bildschirminhalten usw.

Die SAVE-Routine des BASIC-Interpreters gehört nicht gerade zu den komfortabelsten. Und wenn es um das Abspeichern von Grafikseiten, Maschinenprogrammen oder ähnlichem geht, dann versagt sie ganz, weil wir dazu die Start- und Endadresse des abzuspeichernden Bereiches angeben müßten. Aber wie immer in diesen Fällen gibt es auch hier einen Trick, zunächst nur für die DATASETTE, um "künstliche" Files (= Aufzeichnungen auf Band oder Diskette) zu erzeugen.

Wie so oft leistet wieder einmal die Zeropage gute Dienste. Im Bereich der Speicherzellen 170 bis 195 finden wir Zeiger und Register für die Dateiverwaltung.

Das wichtigste sind zunächst die Zeiger für Anfang und Ende des abzuspeichernden Bereiches. Den Zeiger auf den Anfang finden wir in den Speicherzellen 193 und 194, der Endvektor liegt in den Registern 174 und 175. Den SAVE-Befehl können wir mit SYS 62954 aufrufen. Wir müssen aber noch einen Namen mit auf die Reise schicken. Diesen schreiben wir am besten in eine REM-Zeile am Anfang des Programmspeichers. Wo der Filename steht, sagt dem Betriebssystem ein Zeiger in den Bytes 187/188.

Schließlich brauchen wir noch Sekundäradresse, Gerätenummer und die Länge des Filenamens. Am besten, Sie sehen es sich selbst an:

```
10 REM Filename
20 POKE 193, SL: POKE 194, SH: REM Startadresse (Low/High)
30 POKE 174, EL: POKE 175, EH: REM Endadresse (Low/High)
40 POKE 187, PEEK (43) + 6: POKE 188, PEEK (44): REM Zeiger auf
Filename
```

```
50 POKE 183, L: REM Filenamenlänge
60 POKE 186, 1: POKE 185, 0: REM Gerät/Sekundäradr.
70 SYS 62954: REM Aufruf der SAVE-Routine im ROM
```

Solcherart abgespeicherte Programmfiles, Grafikseiten u.ä. können mit LOAD "Filename", 1, 1 an die gleiche Stelle zurückgeladen werden, da die Anfangsadresse mitgespeichert wird.

Noch einige Erläuterungen zu Zeile 40: Hier wird dem Betriebssystem mitgeteilt, wo es den Filenamen findet. Steht dieser in der ersten Zeile hinter REM, so muß man nur 6 zum Zeiger auf den BASIC-Anfang addieren, um die Position zu erhalten. Sollte PEEK (43) + 6 einen größeren Wert als 255 ergeben, so gibt der Rechner einen ILLEGAL-QUANTITY-ERROR aus (dies kann aber nur bei sehr seltenen Speichereinstellungen passieren). In diesem Fall sollte man die Adresse aus der Formel PEEK (43) + 6 + PEEK (44) * 256 berechnen und daraus dann die neuen Zeigerbytes ableiten.

Besitzer einer Floppy-Station haben es erheblich leichter. Sie können nämlich mittels einer sinnreichen BASIC-Funktion fast wie mit POKE auf die Diskette schreiben. Gemeint ist die Befehlskombination PRINT #1, CHR$ (x). Sie schickt genau ein ASCII-Zeichen zur Floppy. Um das sinnvoll anwenden zu können, muß man das Format der Speicherung von Programmfiles auf Diskette kennen. Jedes Programm besteht aus einem Directory-Eintrag und dem Programmtext; am Beginn des Textes stehen zwei Bytes, die die Startadresse für den Ladevorgang angeben. Im Normalfall sind dies 0 und 8 (0 + 256 * 8 = 2048 = BASIC-Start). Der Programmtext ist byteweise als Folge von Interpretercodes gespeichert. Grundsätzlich wird jedes Byte von der Floppy-Station wie ein ASCII-Code behandelt, gleich welchen Zweck es erfüllt. Wenn man also einen Zeiger ausliest, der aus zwei Bytes mit Inhalt 65 und 66 besteht, dann erscheinen diese nach GET #1, A$, B$ als A und B in den beiden Strings. Läßt man sich die ASCII-Codes dieser Strings ausgeben, so erscheinen die beiden Bytes.

Schicken wir ein Zeichen mit PRINT #1, CHR$ (x) zur Floppy, so wird die Zahl x auf der aktuellen Position auf der Platte gespeichert.

Würden wir dagegen PRINT #1, X eingeben, so würde X als mehrere Bytes lange Folge abgelegt (1 Byte pro Stelle).

Daraus ergibt sich ein einfaches Verfahren, um Programmfiles zu erzeugen:

1. Directoryeintrag erzeugen
   Dies übernimmt ein OPEN-Befehl für uns.
2. Startadresse "poken"
   Dies geschieht folgendermaßen:
   PRINT #1, CHR$ (Lowbyte): PRINT #1, CHR$ (Highbyte)
3. Text abspeichern
   Der Text kann zum Beispiel auch eine Bildschirmgrafik sein, die byteweise abgespeichert wird.

Hier das entsprechende Programm, das einen Bildschirminhalt auf Diskette kopiert:

```
10 OPEN 1, 8, 1, "0:BILDSCHIRM"
20 PRINT #1, CHR$ (0);: PRINT #1, CHR$ (4);: REM Startpointer
30 FOR I = 1024 TO 2023
40 PRINT #1, CHR$ (PEEK (I));
50 NEXT I: CLOSE 1
```

Auch hier kann das Ergebnis mit LOAD "BILDSCHIRM", 8, 1 zurückgeladen werden.

Zeile 10 sollte nur im Filenamen (hinter "0:") verändert werden. Ganz wichtig ist die Sekundäradresse 1, die dem DOS (Disk

Operating System = Diskettenbetriebssystem) mitteilt, daß wir SAVEn wollen.

Auf keinen Fall sollte man das CLOSE am Ende der Routine vergessen, da der File sonst zerstört würde!

Soll ein alter File überschrieben werden, so muß vor die Null innerhalb der Anführungsstriche noch ein Klammeraffe gesetzt werden.

4.2. Merge per Hand

Wir kommen jetzt zu einem häufig auftretenden Problem. Oft gibt es Programmteile, die getrennt getestet und abgespeichert wurden und nun zusammen ein wunderbares Paar abgeben könnten. Es bleibt in den meisten Fällen nichts anderes übrig, als einen der beiden Teile neu einzutippen, es sei denn, man besitzt ein MERGE-Hilfsprogramm, mit dem man Programme einfach aneinanderhängen kann. Das muß aber nicht so sein. Mit ein paar einfachen Befehlen kann man dies auch "von Hand" erreichen.

Wenn wir das Problem genauer besehen, so reduziert es sich auf die Tatsache, daß das nachzuladende Programm das alte überschreibt. Es wäre wünschenswert, wenn man dem Interpreter sagen könnte, wohin er den zweiten Teil laden soll. Da dies aber nicht ohne Schwierigkeiten geht, lautet die logische Konsequenz, den Bereich des alten Programms vor dem Interpreterzugriff zu schützen (und das können wir)!

Ist der Speicher erst einmal geschützt, können wir den neuen Programmteil einfach mit LOAD nachladen und danach den geschützten Bereich wieder freigeben. Wichtig ist aber, daß der zweite Teil höhere Zeilennummern als der erste hat. Sonst könnten wir die angehängten Zeilen zwar listen, doch könnte der Interpreter sie nicht ausführen.

Hier nun die genaue Vorgehensweise:

1. PRINT PEEK (43), PEEK (44)

Damit wird der Pointer auf den BASIC-Anfang ausgegeben (im Normalfall 1 und 8). Diese beiden Zahlen müssen wir uns unbedingt merken, da wir damit später die alte Konfiguration wieder herstellen wollen.

2. POKE 44, (PEEK(45) + 256 * PEEK(46) - 2) / 256
   POKE 43, (PEEK(45) + 256 * PEEK(46) - 2) - PEEK(46)
   *256

In den Speicherzellen 45 und 46 befindet sich der Zeiger auf dem Beginn des Variablenbereichs. 2 Bytes vor dem Variablenstart endet das BASIC-Programm.

Der Variablenbereich setzt immer genau nach dem Programmtext an, weil er bei jeder Änderung von Zeilen um ein entsprechendes Stück verschoben wird. Wir verlegen mit den beiden POKE-Befehlen also den Programmanfang (zumindest für den Interpreter) hinter den alten Text und schützen ihn so vor dem Überschreiben.

3. NEW

Da sich die übrigen Zeiger der neuen Situation anpassen müssen und außerdem Reste von Variablen als unerwünschte Programmzeilen interpretiert werden könnten, müssen wir den verbleibenden Speicherbereich mit NEW neu initialisieren. Das ursprüngliche Programm wird davon jedoch nicht berührt (es ist ja jetzt geschützt).

4. LOAD

Jetzt können wir das anzuhängende Programm einfach mit LOAD in den Speicher bringen. Wir dürfen aber auf keinen Fall LOAD "Name", X, 1 eingeben, da dann das ursprüngliche Programm ohne Rücksicht auf die Zeiger überschrieben würde.

Es besteht außerdem jetzt die Möglichkeit, anstelle eines Programms eine Directory zu laden, ohne das im Speicher befindliche Programm zu zerstören. In diesem Falle können wir das Inhaltsverzeichnis nach dem Ladevorgang ganz normal listen. Vor dem nächsten Schritt müßte dann allerdings nochmals NEW eingegeben werden, damit die Directory gelöscht wird (sie soll ja nicht angehängt werden). Nach der ganzen Prozedur erhalten wir dann den Ausgangszustand zurück.

5. POKE 43, 1. gemerkte Zahl: POKE 44, 2. gemerkte Zahl

Hiermit stellen wir die ursprüngliche Speicherkonfiguration wieder her. Die beiden Programme werden dadurch aneinandergehängt und können jetzt zusammen benutzt und abgespeichert werden.

Zusammenfassung: Merge per Hand

1. PRINT PEEK (43), PEEK (44)
   Zahlen merken!
2. POKE 44, (PEEK(45) + 256 * PEEK(46) - 2) / 256
   POKE 43, (PEEK(45) + 256 * PEEK(46) - 2) - PEEK(44) *256
3. NEW
4. LOAD
5. POKE 43, 1. gemerkte Zahl: POKE 44, 2. gemerkte Zahl

4.3. Directories

Dieser Abschnitt bezieht sich leider nur auf das Diskettenlaufwerk VC-1541. Die geneigten Leser ohne dieses nützliche Requisit mögen mir verzeihen und bis zum nächsten Abschnitt weiterblättern.

Den ersten Trick kennen wir schon aus dem letzten Abschnitt, nämlich das Laden einer Directory ohne Programmverlust. Es kann aber auch nützlich sein, das Inhaltsverzeichnis per Programm zu laden und dann z.B. als Array abzulegen

(beispielsweise für Dateiverwaltungsprogramme etc.). Ein Programm hierzu finden Sie im Anhang der 1541-Anleitung und auf der TEST/DEMO-Diskette unter dem Namen "DIR". Man kann es sich für eigene Zwecke leicht umschreiben. Es ist aber auch sehr interessant, sich einmal die Struktur der Directory anzusehen. Auch hierzu liefert das Handbuch gute Informationen. Außerdem sollten Sie sich nicht scheuen, die Directory mittels OPEN 1,8,5,"$" anzusprechen (vorsichtshalber eine Versuchsdiskette nehmen) und per GET#1, A$ byteweise auszulesen.

Es spricht wieder einmal gegen das VC-1541-Handbuch, daß es Floppy-Befehle gibt, die nicht aufgeführt wurden. So kann man das Inhaltsverzeichnis auch nach bestimten Kriterien sortiert laden.

Die einfachste Form ist LOAD "$$",8. Damit wird nur noch der Diskettenname und die Anzahl freier Blocks geladen.

Will man nur bestimmte Einträge ansehen (z.B. alle Files, die mit ABC beginnen), so benutzt man LOAD "$:ABC*",8

Ein ähnliches Verfahren gibt es für Filetypen. Hier heißt der Befehl LOAD "$:*=Typ",8, wobei für "Typ" der Anfangsbuchstabe der Dateiart (Prg, Seq, Rel, Usr) einzusetzen ist. Jetzt wird die Directory zwar noch geladen, aber es erscheinen nur die Files des angegebenen Typs.

Das funktioniert z.B. auch bei SCRATCH (PRINT#15, "S:*=S" löscht alle sequentiellen Files).

Zum Schluß noch ein Trick, der Geld sparen hilft. Normalerweise verarbeitet das Laufwerk nur Disketten vom Typ "singlesided", also einseitig beschreibbar.

Die meisten dieser Single-sided-Disketten sind jedoch auch beidseitig benutzbar, wenn man eine zweite Schreibschutzkerbe anbringt. Dies geht am besten, wenn man einen Locher nimmt, dessen Boden abgenommen wurde. Man kann dann leicht durch die Bohrung eine vorher angebrachte Markierung in Höhe der

anderen Kerbe anpeilen und lochen. Die zusätzliche Kerbe kann kleiner sein als die alte. Es reicht also eine halbmondförmige Lochung, wie sie Abb. 6 zeigt.

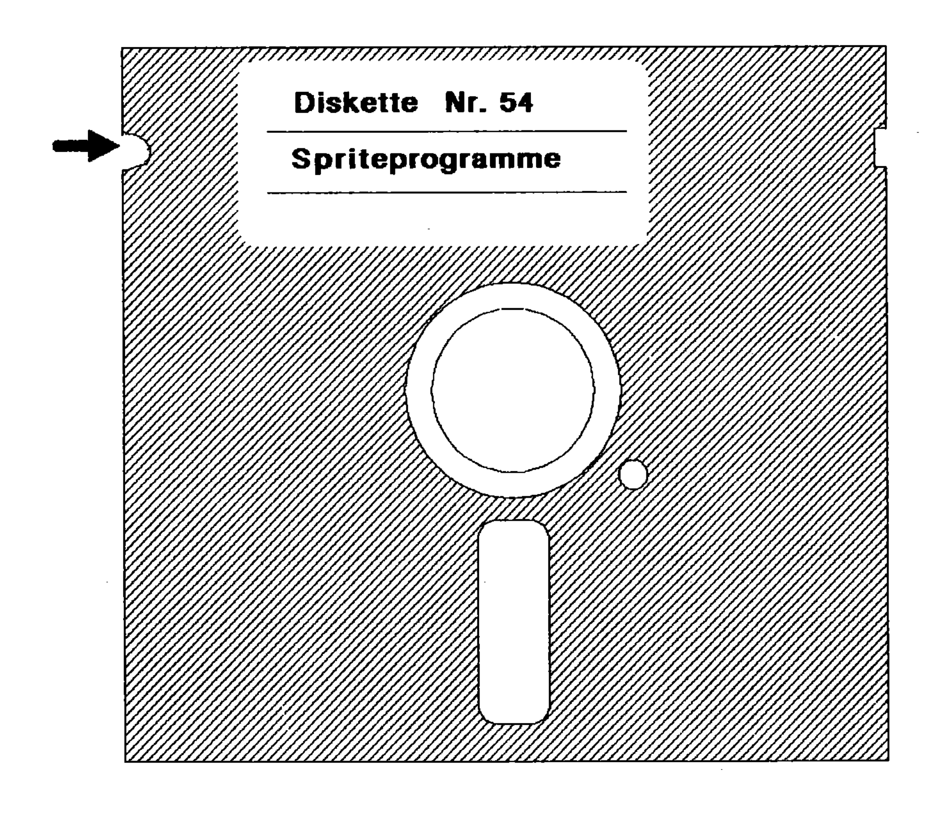


Abb. 6: **Diskettentuning für beidseitige Nutzung**

Um endgültige Gewißheit über die Funktionstüchtigkeit des neuen Speicherplatzes zu erhalten, sollte man das Programm CHECK-DISK von der TEST/DEMO-Diskette einsetzen. Es beschreibt alle Spuren einer formatierten Diskette mit Daten und testet dann auf Lesefehler, die eine fehlerhafte Beschichtung anzeigen. Sollten einzelne Blocks beschädigt sein, so wird dies auf dem Bildschirm angezeigt und die entsprechenden Sektoren werden für weitere Zugriffe gesperrt. Dummerweise weist das Programm aber einige Fehler auf. So dauert der Test einer einzigen Diskette bis zu 2 Stunden, obwohl diese Zeit mit einer winzigen Änderung auf 10 Minuten gekürzt werden kann. Ändern Sie einfach die GOTO-Befehle in den Zeilen 150 und 170 so ab, daß der Sprung in die Zeile 90 führt. Außerdem muß noch folgende Zeile eingefügt werden:

```
213 PRINT #15, "V"
```

Jetzt sollte das Programm zufriedenstellend arbeiten.

Zusammenfassung: Directories

LOAD "$$",8 lädt nur Header und Blockanzahl.

LOAD "$:ABC*",8 lädt Directory nur mit Files, deren Name mit ABC beginnt.

LOAD "$:*=Typ",8 lädt Directory nur mit Files, die vom angegebenen Typ sind.

4.4. Verschiedenes rund um die Peripherie

Nach den "großen" Tricks nun ein paar kleine PEEKs und POKEs, die bei der Programmierung der Datenein- und -ausgabe helfen können.

Es kann nützlich sein, die Anzahl der bereits offenen Files zu erfahren. Wie Sie wissen, dürfen maximal 10 Files gleichzeitig geöffnet sein. Wird ein elfter eröffnet, so reagiert der Rechner

mit einem TOO-MANY-FILES-OPEN-ERROR. Diesen kann man vermeiden, wenn man sich vorher mittels PEEK (152) die Anzahl der offenen Dateien ausgeben läßt.

Mit dem CMD-Befehl kann man - wie bekannt - die Ausgabe vom Bildschirm auf Peripheriegeräte umleiten. Die physikalische Adresse dieses Gerätes steht in der Speicherzelle 154 (1 = Datasette, 4 = Drucker, 8 = Floppy). Ebenso kann die CMD-Belegung durch POKE 154, 3 wieder rückgängig gemacht werden. Ein File bleibt davon unberührt. Deshalb kann die Ausgabe auch durch POKE 154, X wieder auf das Gerät umgeleitet werden.

Ähnlich funktioniert Speicherzelle 153. Hier wird das aktuelle Eingabegerät gespeichert. Soll der Computer z.B. über eine V.24-Schnittstelle ferngesteuert werden, so könnte hier eine 2 stehen. Erhält der Rechner Daten von einem Peripheriegerät, so ist die entsprechende Gerätenummer gespeichert, bei normalem Tastaturbetrieb ist es eine 0.

Ebenfalls interessant ist Speicherzelle 184. Hier steht die Nummer des zuletzt verwendeten Files. Die Sekundäradresse dieses Files steht in Register 185. Hier kann man z.B. feststellen, ob ein Drucker in einen bestimmten Modus gebracht wurde o.ä.

Der Dritte im Bunde ist Speicherzelle 186. Hier steht die physikalische Nummer des zuletzt benutzten Geräts. Diese Adresse kann man in Programmen benutzen, die je nach Ausstattung des gerade verwendeten Computers entweder auf Kassette oder Diskette zugreifen sollen. Nach dem Laden stellt das Programm dann anhand der Speicherzelle 186 fest, welches Gerät der jeweilige Anwender besitzt (sprich: woher das Programm geladen wurde) und kann dann dementsprechend reagieren.

Interessant könnte auch Register 147 sein. Hier läßt sich feststellen, ob der letzte Lesebefehl für Floppy bzw. Datasette ein LOAD (=0) oder ein VERIFY (=1) gewesen ist.

Der letzte Trick wendet sich an Besitzer einer Datasette. Speicherzelle 150 enthält den Kassettenmotorflag. Ist der Motor

nicht eingeschaltet, so enthält dieses Byte 0, andernfalls ist es ungleich 0.

Zusammenfassung: Peripherietricks

PRINT PEEK (152): Anzahl offener Files
PRINT PEEK (153): aktuelles Eingabegerät
PRINT PEEK (154): aktuelles Ausgabegerät
PRINT PEEK (184): aktueller File
PRINT PEEK (185): aktuelle Sekundäradresse
PRINT PEEK (186): aktuelles Gerät
PRINT PEEK (147): letzter Lesebefehl
PRINT PEEK (150): Kassettenmotorflag

4.5. Die Statusvariable ST

Sie haben sicher schon von der Statusvariablen ST gehört. Sie zeigt Fehler bei LOAD und VERIFY vom Band und bei der Benutzung des seriellen Busses an.

Je nach Art des Fehlers werden verschiedene Bits der Variablen gesetzt. Ist kein Fehler aufgetreten, so ist ST= 0. Beim seriellen Bus wird die Meldung DEVICE NOT PRESENT durch den Wert -128 angezeigt. Sollte ein Schreibfehler aufgetreten sein, so ist ST= 1, beim Lesen ist ST= 2. Sollte das Ende eines Datenblocks erreicht sein, so ist der Wert von ST 64 (sowohl bei Kassette als auch bei Diskette).

Das Bandende (bei Datasette) wird durch -128 angezeigt, ein Prüfsummenfehler durch 32. Dieser Fehler kann auch dann aufgetreten sein, wenn die Operation ordnungsgemäß und ohne Fehleranzeige abgeschlossen wurde. Konnte der Fehler aber nicht "ausgebügelt" werden (durch den Kontrollblock bei LOAD und VERIFY), so ist Bit 4 (=16) gesetzt. Sollten sich Fehler in der Blocklänge ergeben, so ist ST= 4 (zu kurz) bzw. 8 (zu lang).

Sind mehrere Fehler gleichzeitig aufgetreten, so wurden die entsprechenden Bits gesetzt und dadurch die Zahlen addiert.

Sollte ein Prüfsummenfehler aufgetreten und außerdem das Bandende erreicht sein, so ist ST= -96 = -128 + 32.

Auf diese Art und Weise lassen sich Fehler bei Band- und Floppyzugriff leicht feststellen.

5. Der Bildschirm

Hier soll vom normalen Bildschirmaufbau und seiner Manipulation die Rede sein, denn es muß nicht immer hochauflösende Grafik sein, mit der ein gutes Bild programmiert werden kann.

5.1. Blockgrafik

Haben Sie sich die Grafikzeichen Ihres 64ers schon einmal genauer angesehen? Es befinden sich darunter auch solche, die genau ein Viertel des Platzes einnehmen, den ein Bildschirmzeichen maximal beansprucht. Ebenso gibt es solche, die genau die Hälfte einnehmen. Nehmen wir den reversen Satz noch dazu, dann haben wir auch Dreiviertel- und ganze Zeichen.

Da der Bildschirm 25 Zeilen mit je 40 Zeichen hat, könnten mit dieser Viertelpunktgrafik 50 x 80 Punkte benutzt werden.

Da die angesprochenen Grafikzeichen praktischerweise zusammen mit der Commodoretaste erreicht werden, können wir sie auch im Kleinschriftmodus benutzen.

Es wäre nur wünschenswert, könnten wir diese Zeichen über ein Unterprogramm wie Grafikpunkte setzen und löschen lassen. Größtes Problem: Wenn schon Punkte gesetzt worden sind, kann man nicht einfach ein anderes Viertelpunktzeichen in die betrefffende Bildschirmzelle POKEn. Dadurch würde der alte Punkt gelöscht. Vielmehr muß das alte Zeichen in das neue mit eingerechnet werden. Je nach Aussehen der alten Bildschirmzelle muß ein bestimmter Code in das TV-RAM gePOKEd werden.

Eine Möglichkeit wäre, für jeden möglichen Punkt einen Platz in einer 50x80-Matrix anzulegen, und in dieser Matrix einfach einzelne Punkte zu setzen oder löschen. Eine Unterroutine rechnet dann jeweils vier Punkte zu einem Bildschirmzeichen zusammen.

Einfacher ist es, für jede mögliche Kombination von Bildschirmzeichen und zu setzendem Punkt in einer Tabelle das zu pokende Byte aufzuschreiben.

Diese Tabelle wird in den Speicher des BASICs übertragen. Dann kann der Rechner sich bei jedem Aufruf der Blockgrafikroutine das gewünschte Zeichen aus der Tabelle heraussuchen. Unten finden Sie ein Programm, daß diese Block- oder Viertelpunktgrafik erzeugen kann.

Im ersten Teil geschieht die Initialisierung, die die nötigen Tabellen einliest. Diese sind recht umfangreich geworden, da auch eine Löschroutine integriert worden ist.

Ab Zeile 60000 stehen die Setz- und Löschunterprogramme. Mit GOSUB 60000 wird die Setzroutine, mit GOSUB 60001 die Löschroutine aufgerufen.

Ist die Variable L gleich 0 (Zeilen 60000, 60001), so wird der erste Teil der Tabelle benutzt (=setzen), bei L=1 wird der zweite Teil benutzt (=löschen).

Die Koordinaten des anzusprechenden Punktes werden in den Variablen X (0 bis 49) und Y (0 bis 79) übergeben. Der Code der Farbe, die benutzt werden soll, steht in CO. Hier das Listing:

```
10 PRINT CHR$(147):CO= 14: DIM P2% (15), PD% (1,1,1,15)
20 FOR B= 0 TO 1: FOR X= 0 TO 1: FOR Y= 0 TO 1: FOR Z= 0 TO 15
30 READ PD% (B,X,Y,Z): NEXT Z, Z, X, B
40 FOR X= 0 TO 15: READ P2% (X): NEXT X
50 DATA 126, 126, 226, 97, 127, 97, 251, 226, 252, 127, 236, 160,
252, 251, 236, 160
51 DATA 123, 97, 255, 123, 98, 97, 254, 236, 98, 252, 255, 254,
252, 160, 236, 160
52 DATA 124, 226, 124, 255, 225, 236, 225, 226, 254, 251, 255,
254, 160, 251, 236, 166
53 DATA 108, 127, 225, 98, 108, 252, 225, 251, 98, 127, 254, 254,
252, 251, 160, 160
```

```
54 DATA 32, 32, 124, 123, 108, 123, 225, 124, 98, 108, 255, 254,
98, 225, 255, 254
55 DATA 32, 126, 124, 32, 108, 126, 225, 226, 108, 127, 124, 225,
127, 251, 226, 251
56 DATA 32, 126, 32, 123, 108, 97, 108, 126, 98, 127, 123, 98,
252, 127, 97, 252
57 DATA 32, 126, 124, 123, 32, 97, 124, 226, 123, 126, 255, 255,
97, 226, 236, 236
58 DATA 32, 126, 124, 123, 108, 97, 225, 226, 98, 127, 255, 254,
252, 251, 236, 160
60000 L= 0: GOTO 60010
60001 L= 1
60010 Y= 49-Y: S= INT (X/2): Z= INT (Y/2): PO= S+40*Z
60020 X1= X-2*S: X2= Y-2*Z: X3= PEEK (1024+PO)
60030 F= 0: FOR I= 0 TO 15: IF X3= P2% (I) THEN X3= I: F= 1
60040 NEXT I
60050 IF F= 0 THEN X3= 0: IF L= 1 THEN RETURN
60060 POKE 1024+PO, PD% (L,X1,X2,X3): POKE 55296+PO, CO: RETURN
```

In den Zeilen 60010 und 60020 werden Position im Bildschirmspeicher (PO) und Art des Punktes innerhalb des Zeichenrasters (links/rechts in X1, oben/unten in X2) berechnet. Die Zeilen 60030 und 60040 suchen anhand des schon vorhandenen Zeichens aus dem Bildschirmspeicher nach der richtigen Tabellenspalte (X3). Sollte das Zeichen kein Graphikzeichen sein, so wird dies in Zeile 60050 registriert. Beim Setzmodus wird das alte Zeichen dann einfach überschrieben, beim Löschen bleibt es zur Freude des Benutzers stehen.

Die Krönung des Unterprogrammes ist Zeile 60060, in der das Zeichen aus der Tabelle herausgesucht und gePOKEd wird.

Ein Wort noch zu den Koordinaten. Der Ursprung des Koordinatensystems ist in der linken unteren Ecke. Damit ist es besonders einfach, Funktionen o.ä. auf den Bildschirm zu bringen.

5.2. Balkengraphik

Für die graphische Darstellung von Bilanzen oder Messwerten ist es nützlich und üblich, Balkengraphiken zu verwenden. So lassen sich zum Beispiel Verkaufszahlen sehr gut in waagerechten Balken für jeden Monat des Jahres verewigen. Leider bietet kaum ein Computer standardmäßig einen Befehl zur Erstellung dieser Balken. Deshalb habe ich unten eine Unterroutine aufgelistet, die die Erzeugung von Balkengraphiken sehr erleichtern kann. Sie wird ähnlich wie die Blockgrafikroutine benutzt.

Auch hier leisten uns die Graphikzeichen wieder gute Dienste. Wir können in horizontaler Richtung 320 verschiedene Längen von Balken ausgeben lassen, da wir auf 40 Zeichen zu je 8 Punkten Breite zugreifen können. Für jede Breite (von 0 bis 8) gibt es ein Grafikzeichen im Vorrat des 64ers.

Wir müssen also nur noch berechnen, wie viele ganze reverse Leerzeichen sich in der Länge des Balkens unterbringen lassen und für den Rest das entsprechende Grafikzeichen auswählen. Auch hierzu wird wieder ein kleines Array benutzt. Hier das Listing:

```
10 DIMXA$(7):FORI=0TO7:READXA$(I):NEXT
20 DATA "  ","| ","| ","▌ ","▌ ","▨ ▀","▨ ▀","▨ ▀"
60500 YM=320-V*8:AN$="":IFY<YMTHENY=YM
60510 XA=Y/8:G=INT(XA):XA=(XA-G)*8
60520 IFG>0THENFORI=1TOG:AN$=AN$+"▨ ▀":NEXTI
60530 AN$=AN$+XA$(XA)
60540 C1=PEEK(214):C2=PEEK(211):C3=PEEK(646)
60550 POKE646,C0:POKE214,X:POKE211,V:SYS58732:PRINTAN$
60560 POKE646,C3:POKE214,C1:POKE211,C2:SYS58732:RETURN
```

Die Bedeutung der ersten beiden Zeilen dürfte klar sein, hier werden die benötigten Zeichen eingelesen. Sicherheitshalber hier

noch einmal die ASCII-Codes der Zeichen aus Zeile 20 (ohne RVS-ON/ RVS-OFF):

32, 165, 180, 181, 161, 182, 170, 167

Aufgerufen wird die Routine mit GOSUB 60500. Die Länge des Balkens (1 bis 320) soll in der Variablen Y übergeben werden. Die Zeile, in der er stehen soll, wird in X angegeben (0 bis 39). Um das Erstellen von Graphiken oder ähnlichem zu erleichtern, kann auch die Spalte bestimmt werden, ab der der Balken starten soll (0 - 24). Sollte die Länge des Balkens den noch freien Raum in der Zeile übersteigen, so wird der Balken automatisch verkürzt. Dies bewirkt Zeile 60500. Zeile 60510 berechnet die Anzahl der ganzen Revers-Zeichen im Balken (G) und die Anzahl der verbleibenden Punkte (XA). Zeile 60520 fügt die ganzen Zeichen zu einem String (AN$) zusammen, Zeile 60530 hängt das letzte Zeichen mit dem Rest an.

Um die normalen PRINT-Befehle nicht zu beeinflussen, wird die alte Cursorposition sowie die gerade benutzte Farbe gespeichert (C1, C2, C3, siehe Zeile 60540). In der nächsten Zeile wird dann der Cursor neu gesetzt, die Farbe Ihren Wünschen entsprechend geändert (=CO) und der Balken ausgegeben. Die letzte Zeile stellt schließlich wieder den alten Zustand her und beendet das Unterprogramm.

Da wieder nur Zeichen benutzt wurden, die auch im Kleinschriftmodus zur Verfügung stehen, kann diese Routine in fast jeder Anwendung eingesetzt werden.

5.3. Die Betriebsarten im Zeichenmodus

In diesem Abschnitt klären wir die Frage, woher die kleinen und großen As, Bs, Cs u.s.w. herkommen. Die Zahl 1, die vielleicht im Video-RAM in Speicherzelle 1024 gespeichert ist, sagt uns zwar, daß an der betreffenden Stelle ein A auf dem Bildschirm erscheinen sollte, doch über die Form dieses Buchstabens sagt sie nichts aus. Das Muster für das A (und alle seine Ge-

schwister vom B bis zum letzten Grafikzeichen) ist im Charakter-ROM gespeichert. Es liegt im Bereich von 53248 bis 57343.

Jedes Bildschirmzeichen beansprucht von diesen 4 Kilobytes genau 8 Bytes, da die Zeichenmatrix 8 x 8 Punkte umfaßt. Jeweils eine Punktzeile belegt ein Byte; ein Bit repräsentiert also einen Punkt. Ist dieses Bit auf 1 gesetzt, so erscheint auf dem Bildschirm ein Punkt in der Farbe, die im Color-RAM an der gleichen Stelle steht; ist das Bit auf 0, so wird dieser Punkt der Matrix die Farbe des Hintergrundregisters 53280 annehmen.

Die Zahl, die im Video-RAM steht, hat dabei eine besondere Aufgabe. Sie wird vom VIC (Video Interface Chip) mit 8 multipliziert und ergibt so die Stelle innerhalb des Charaktergenerators, ab der das gewünschte Muster zu finden ist. Versuchen Sie es einmal. Nehmen Sie ein beliebiges Zeichen , suchen Sie dessen Bildschirmcode aus der Tabelle im CBM-Handbuch heraus (nicht ASCII-Code) und starten Sie dann folgendes Miniprogramm:

```
1 DIM M(7)
5 PRINT CHR$(147)
10 INPUT "Bildschirmcode"; C
20 AD= 53248 + C * 8
30 POKE 56334, PEEK (56334) AND 254
40 POKE 1, PEEK (1) AND 251
50 FOR I= 0 TO 7: M(I)= PEEK (AD + I): NEXT I
60 POKE 1, PEEK (1) OR 4
70 POKE 56334, PEEK (56334) OR 1
80 FOR I= 0 TO 7: FOR J= 7 TO 0 STEP -1
90 IF (M(I) AND 2 ^ J) THEN PRINT "*";: GOTO 110
100 PRINT ".";
110 NEXT J: PRINT: NEXT I
120 PRINT "TASTE": POKE 198,0: WAIT 198,1: GET A$: GOTO 5
```

Mit diesem Programm können Sie sich die Bitmuster der Zeichen im Großschriftmodus ansehen. Wollen Sie die Zeichen des Kleinschriftmodus ausgeben lassen, so muß die Basisadresse in

Zeile 20 von 53248 in 55296 geändert werden. Andere Möglichkeit: Sie addieren einfach 256 zum Bildschirmcode (Beispiel: a = 1 + 256 = 257).

Bitte wundern Sie sich nicht über die kleinen Tricks im Programm, die Sie noch nicht kennen; sie werden in späteren Abschnitten vorgestellt.

Es reicht zur Erklärung, daß sich das "Prográmmchen" in zwei Teile trennen läßt. Im ersten Teil (Zeilen 5 - 70) werden die gewünschten Bytes des Zeichengenerators in das Array M(I) eingelesen. Dazu wird der Interrupt abgeschaltet (Zeile 30) und das Charaktergenerator-ROM eingeschaltet (Zeile 40). Es folgen die FOR-NEXT-Schleife mit den PEEKs (AD enthält übrigens die Startadresse des gewählten Zeichens) und das Wiedereinschalten von I/O-Bereich und Interrupt.

Im zweiten Teil werden die acht Bits in Bitmuster zerlegt. Wenn das n-te Bit in M(I) auf 1 ist, so ergibt der Ausdruck (M(I) AND 2 ^ n) eine 1. Dies nimmt der IF-Befehl zum Anlaß, in den THEN-Teil zu verzweigen und einen Stern auszugeben. Andernfalls erscheint ein Punkt (Zeile 100). Das Besondere daran ist, daß im IF-THEN kein Vergleich steht, sondern lediglich ein Term in Klammern. Sobald dieser ungleich 0 wird, verzweigt der Interpreter zum THEN. Statt des Ausdrucks könnte man auch eine Variable oder ähnliches setzen.

Nun aber zurück zum eigentlichen Thema! Wie Sie der Überschrift dieses Abschnitts vielleicht entnehmen, ist der beschriebene Modus nicht der einzige.

Der EXTENDED-COLOR-MODE ist dem Normalmodus recht ähnlich. Die Bits des Zeichenmusters, die auf 1 gesetzt sind, ergeben nach wie vor einen Punkt in der im Color-RAM gespeicherten Farbe. Die Farbe der 0-Bits kann dagegen verschieden sein. Sie richtet sich nach den Registern für die Hintergrundfarben 0 - 3 (53281 - 53284). Einigen Lesern wird sich jetzt vielleicht ein lautes "Aha!" entringen, da diese Register im Anhang

des CBM-Handbuchs zwar aufgeführt sind, ihre Anwendung aber nicht beschrieben wurde.

Welche der 4 Farben für die 0-Bits verwendet wird, richtet sich nach den beiden höchstwertigen Bits des Bildschirmcodes im Video-RAM. Betrachtet man diese beiden als eigenständige Zahl (Zahl = Bit 7 * 2 + Bit 6), so gibt diese die Nummer des verwendeten Registers an (Beispiel: 1 0 = 1 * 2 + 0 = 2).

Diese beiden Bits lassen sich nun allerdings nicht mehr als Zeiger auf die Position im Charaktergenerator einsetzen. Dafür bleiben nur noch 6 Bits übrig. Damit lassen sich dann die ersten 2 ^ 6 = 64 Zeichen ansprechen.

Den Extended-Color-Mode können Sie durch POKE 53265, PEEK (53265) OR 64 an- und durch POKE 53265, PEEK (53265) AND 191 wieder einschalten.

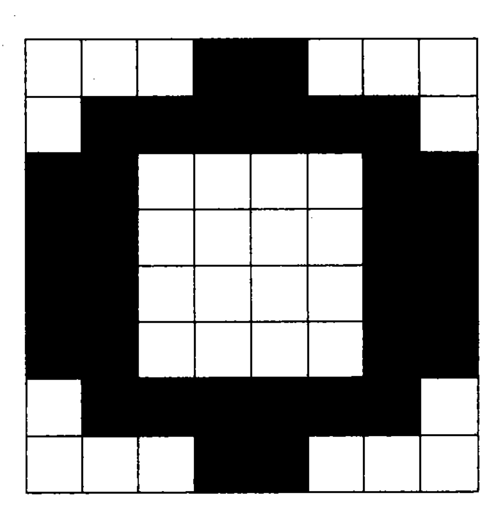
Jetzt wird es haarig! Der Multi-Color-Mode ist ziemlich kompliziert, kann aber bei richtiger Anwendung tolle Ergebnisse liefern.

Wie Sie sich entsinnen werden, kann jedes Bildschirmzeichen höchstens zwei Farben haben: Zeichenfarbe (aus dem Color-RAM) und Hintergrundfarbe (aus den VIC-Registern). Der Multi-Color-Mode erlaubt dagegen bis zu 4 Farben pro Zeichen. Erkauft wird dies allerdings mit einer Vereinfachung der Punktmatrix.

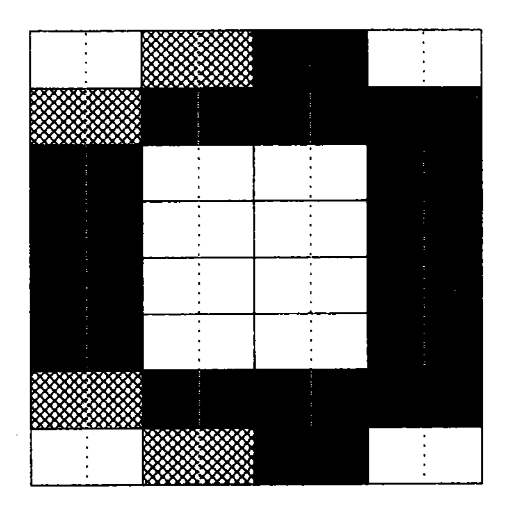
Verantwortlich ist diesmal das Farbbyte aus dem Color-RAM. Ist Bit 3 nicht gesetzt (Byte (AND 2 ^ 3) = 0), dann bleibt fast alles beim alten. Leider können jetzt aber nur noch die Farben 0 bis 7 benutzt werden, da ja das für mehr Farben nötige Bit 3 durch das Multi-Color-Flag belegt ist.

Ist das Bit 3 jedoch auf 1, so wirkt die Mehrfarbigkeit (endlich). Die normale 8 x 8 Matrix wird in eine 4 x 8 Matrix umgewandelt (siehe Abb. 7). Je zwei Bits des Charactergenerators werden jetzt zu einem Punkt zusammengefaßt. Sind beide Bits auf 0, so erhält dieser Punkt die Hintergrundfarbe. Sind beide Bits auf 1,

so holt sich der VIC die Farbe aus dem Color-RAM (allerdings wieder nur die Farben 0 bis 7). Bei den anderen beiden Kombinationen (0 1 bzw. 1 0) wird die Farbe für den Punkt wie beim Extended-Color-Mode aus den Hintergrundfarbenregistern 1 und 2 geholt. Auch diesen Modus können Sie vom BASIC aus per POKE 53270, PEEK (53270) OR 16 einschalten. Ausgeschaltet wird mit POKE 53270, PEEK (53270) AND 239.



Normalmodus



Multicolormodus

Abb. 7: Vereinfachte Matrix im Multicolormodus

Wie Sie jetzt sehen, sind die Zeichen in diesem Modus sehr zerfranst und chaotisch. Findige Programmierer können sich aber den Charaktergenerator ins RAM kopieren und dann "vernünftige" Mehrfarbenzeichen entwerfen.

Zusammenfassung: Betriebsarten im Zeichenmodus

Extended-Color-Mode an: POKE 53265, PEEK (53265) OR 64
Extended-Color-Mode aus: POKE 53265, PEEK (53265) AND 191
Multi-Color-Mode an: POKE 53270, PEEK (53270) OR 16
Multi-Color-Mode aus: POKE 53270, PEEK (53270) AND 239

5.4. Character-Generator verlegen

Wie wohl unschwer zu erraten ist, wird auch die Lage des Zeichengenerators vom VIC gesteuert (er ist der Tausendsassa des C-64; er überwacht die Zusammenarbeit von Prozessor und übrigen Bausteinen, erzeugt ein Videosignal, steuert Sprites und hochauflösende Grafiken und generiert so ganz nebenbei auch den Takt für den gesamten Rechner). Ganz speziell sollte uns hier die Speicherzelle 53272 interessieren.

Innerhalb dieser Speicherzelle bestimmen die Bits 1 bis 3 die Adresse des Zeichengenerators. Diese hängt zwar noch von anderen Faktoren ab, doch diese sind schwerer zu beeinflussen. Alle folgenden Angaben sind deshalb auf die Normalkonfiguration zugeschnitten; etwaige Hilfsprogramme können diese unter bestimmten Umständen verändern (vor allem Grafikhilfsprogramme).

Die untenstehende Tabelle zeigt, welche Bitkombinationen welche Bereiche adressieren.

```
000 -->     0
001 -->  2048
010 --> Großschrift
011 --> Kleinschrift
100 -->  8192
101 --> 10240
110 --> 12288
111 --> 14336
```

Eine Sonderstellung nehmen dabei die Kombinationen 010 und 011 ein. Sie adressieren das ROM (53248 bzw. 55296).

Auf die 3 Bits in Speicherzelle 53272 sollte am besten nur per AND und OR zugegriffen werden. Um die Belegung zu ändern, sollte man das Byte zuerst mit der Binärzahl 1111 0001 (=241) AND-verknüpfen. Dadurch werden die Bits 1 bis 3 gelöscht. Sodann können per OR die gewünschten Kombinationen eingestellt werden. Beispiel: Um den Character-Generator nach 2048 (BASIC-Anfang!) zu verlegen, muß die Kombination 001 in Speicherzelle 53272 stehen. Bit 0 dieses Bytes kann von uns nicht beeinflußt werden, es bleibt immer auf 1. Wir bilden daher von der Bitkombination die dezimale Entsprechung; das ist in unserem Fall 1. Da das Ganze um ein Bit nach links verschoben im Byte stehen soll, muß noch mit 2 multipliziert werden. Der gesamte Befehl zum Verschieben des Zeichengenerators lautet also:

```
POKE 53272, (PEEK (53272) AND 241) OR 2
```

Damit ist der Generator zwar verlegt, doch wir können damit nichts anfangen, auf dem Bildschirm steht jedenfalls nur ein Punktegewirr. Machen wir uns also ans Werk. Mit dem untenstehenden "Prögrämmchen" kann das Charactergenerator-ROM ausgelesen und ins RAM kopiert werden. Vorher müssen wir allerdings noch den BASIC-Anfang nach 6144 verlegen, da ja die

ersten 4 K vom Generator belegt werden sollen. Dies geschieht durch

POKE 43,1: POKE 44,24: POKE 6144,0: CLR.

Soll ein Programm selbsttätig den Zeichengenerator verlegen, so muß ein Ladeprogramm benutzt werden; siehe Kapitel 3.3.

Jetzt können die unten aufgeführten Zeilen eingegeben werden:

```
10 POKE 56334, PEEK (56334) AND 254: REM Interrupt aus
20 POKE 1, PEEK (1) AND 251: REM ROM einschalten
30 FORI= 0 TO 4095: POKE 2048+I, PEEK (53248+I): NEXTI
40 POKE 1, PEEK (1) OR 4: REM ROM ausschalten
50 POKE 56334, PEEK (56334) OR 1: REM Interrupt ein
```

Wenn Sie jetzt auf den neuen Charaktergenerator umschalten, haben die Zeichen immer noch ihre alte Form, doch - und das ist das Erfreuliche - kommt diese nun aus dem RAM. Dort läßt sie sich leicht per POKE ändern. Bildschirmzeichen lassen sich wie Sprites definieren, unterschiedlich sind nur die Matrix (8 x 8 statt 21 * 24) und die Lage im Speicher.

Zum Betrachten der Muster können Sie wieder das Programm aus Kapitel 5.2. benutzen, doch das Abschalten des Interrupts und das Verändern von Speicherzelle 1 ist jetzt unnötig. Natürlich muß auch die Basisadresse in Zeile 20 (jetzt 2048) verändert werden.

Die neuen Bildschirmzeichen können Sie jetzt einfach einpoken - versuchen Sie es! Besonders im Multi-Color-Mode sind der Kreativität des Programmierers keine Grenzen gesetzt.

Abschalten können Sie den neuen Zeichensatz übrigens mit POKE 53272, (PEEK (53272) AND 241) OR 4 (für Großschrift) bzw. 6 (für Kleinschrift).

Zusammenfassung: Verlegen des Zeichengenerators

Bits 1 - 3 der Speicherstelle geben den Ort des Generators an. Bei Verlegung in den BASIC-Speicherbereich ist ein Schützen des belegten Bereiches notwendig (siehe Kap. 3.3.). Zeichensatz kann nach Abschalten des Interrupts und ROM-Umschaltung ausgelesen und ins RAM gePOKEd werden.

Eigener Satz ein: POKE 53272, (PEEK (53272) AND 241) OR x

Eigener Satz aus: POKE 53272, (PEEK (53272) AND 241) OR 4 bzw. 6

5.5. Video-RAM verlegen

Ähnlich wie der Charactergenerator läßt sich auch das Video-RAM durch die Speicherzelle 53272 verschieben. Zuständig sind diesmal die Bits 4 bis 7.

Mit diesen vier Bits läßt sich das TV-RAM in Schritten von einem Kilobyte verschieben. Im Normalfall ist nur Bit 4 gesetzt, was dann den Bereich von 1024 bis 2023 selektiert. Hier wieder eine Tabelle mit Bitkombinationen und deren Ergebnis:

```
0000 -->     0
0001 -->  1024
0010 -->  2048
0011 -->  3072
0100 --> ROM
0101 --> ROM
0110 --> ROM
0111 --> ROM
1000 -->  8192
1001 -->  9216
1010 --> 10240
1011 --> 11264
1100 --> 12288
```

```
1101 --> 13312
1110 --> 14336
1111 --> 15360
```

Wie Sie sehen, bilden die Kombinationen, die mit ROM gekennzeichnet sind, eine Ausnahme. Dies ist notwendig, damit der VIC auf den Charaktergenerator im ROM zugreifen kann. Diese Bereiche werden in den Speicher von 4096 bis 8191 hineingespiegelt. Für den VIC liegt das ROM also hier und nicht ab 53248! Umgeschaltet wird wieder mit AND, OR, PEEK und POKE. Zunächst müssen die vier höchstwertigen Bits gelöscht werden. Das geht am besten durch AND 15. Sodann müssen wir die Binärkombination in eine Dezimalzahl verwandeln. Wollen wir das TV-RAM nach 15360 verlegen, so wäre dies 15. Diese Zahl muß (wegen der Verschiebung im Byte) mit 16 multipliziert werden. Das Ergebnis setzen wir in die OR-Verknüpfung ein. Der gesamte Befehl lautet dann:

POKE 53272, (PEEK (53272) AND 15) OR X.

Ist Ihnen an der Zahl, die wir vor der Multiplikation aus der Binärkombination errechnet hatten, etwas aufgefallen? Richtig - sie gibt an, im wievielten Kilobyte des Speichers das Video-RAM stehen soll. In Zukunft brauchen Sie also nicht mehr mühsam Binärkombinationen umrechnen, Sie setzen einfach das gewünschte K in den folgenden Befehl ein:

POKE 53272, (PEEK (53272) AND 15) OR K * 16.

Eine arge Enttäuschung erleben Sie aber, wenn Sie diesen Befehl so ausprobieren. Auf dem Bildschirm erscheint ein heilloses Durcheinander von Zeichen, die sich auch über die Tastatur nicht unbedingt bändigen lassen. Wir haben vergessen, dem Betriebssystem mitzuteilen, wo das neue TV-RAM steht. Der VIC holt sich jetzt die Bildschirmdaten von einem Ort, wo das Betriebssystem noch gar nichts ablegt. Wenn Sie in dieser Situation die Taste CLR drücken, so löscht das Betriebssystem noch den alten Bildschirmspeicher und das Color-RAM. Aber auch dagegen ist ein Kraut gewachsen.

Die Speicherzelle 648 teilt dem Rechner das Highbyte der Video-RAM-Startadresse mit. Dieses erhalten wir, indem wir die Startadresse durch 256 teilen. Um bei unserem Beispiel zu bleiben: 15360 / 256 ergibt 60. Mit POKE 648, 60 ist die Welt dann für uns und den C-64 wieder in Ordnung. Zum Glück gibt es auch hier wieder eine Vereinfachung. Es reicht aus, die Kilobyte-Nummer mit 4 zu multiplizieren, um das gewünschte Highyte zu bekommen.

Noch etwas ist zu beachten, wenn Sie Sprites verwenden. Die Zeiger auf die Blöcke, in denen die Sprites definiert werden, liegen nicht mehr in den Speicherzellen 2040 bis 2047. Sie wurden mitverschoben. In unserem Beispiel lägen sie also im Bereich von 16376 bis 16383.

Bitte denken Sie auch daran, daß wieder der BASIC-Speicherbereich geschützt werden muß, wenn Sie das Video-RAM verlegen.

Besonders reizvoll erscheint mir die Möglichkeit, zwei getrennte Bildschirmseiten zu definieren und zwischen beiden bei Bedarf hin- und herzuschalten. Allerdings wirken die PRINT-Befehle des BASICs nur auf die gerade eingeschaltete Seite, auf die andere müßte dann mit PEEK und POKE zugegriffen werden. Weiteres Problem: Das Color-RAM kann nicht verschoben werden, beide Bildschirmseiten müssen also die gleichen Zeichenfarben benutzen.

Zusammenfassung: Video-RAM verlegen

TV-RAM kann über die Bits 4 - 7 der Speicherzelle 53272 gesteuert werden. Die Nummer des Kilobytes, das den Bildschirmspeicheraufnehmen soll, ist in K einzusetzen:

```
POKE 53272, (PEEK (53272) AND 15) OR K * 16
POKE 648, K * 4
```

5.6. Verschiedene Tricks für den Bildschirm

Auch für den normalen Zeichenmodus gibt es einige Tricks, die einem das Programmieren einer Textausgabe o.ä. erleichtern können.

Fangen wir mit der Farbe an. Die zur Zeit eingeschaltete Zeichenfarbe können Sie aus Speicherzelle 646 erfahren. Mit POKE 646, Farbcode können Sie das gleiche bewirken wie über die Tastenkombinationen CTRL + Farbe bzw. Commodore + Farbe. Diese Methode bietet aber den Vorteil, daß der Farbcode direkt angegeben werden kann. Dies ist zum Beispiel nützlich, wenn man die Schriftfarbe je nach RND-Wert zufällig umschalten will.

Die Speicherzelle 647 nennt überdies die Farbe des Zeichens unter dem Cursor (auch, wenn dieser abgeschaltet ist). Diese läßt sich aber leider nicht durch POKE 647, X verändern.

Apropos verändern: Für alle Speicherzellen, die mit Farben zu tun haben, gilt, daß sich die Bits 4 bis 7 verändern können. Dies sollte uns aber nicht stören, da die Farbencodes ja doch nur von 0 bis 15 reichen und demnach nur die Bits 0 - 3 belegen. Wundern Sie sich deshalb nicht, wenn die Speicherzelle 55296 im Color-RAM, die Sie eben mit 0 "gefüllt" haben, plötzlich eine 32 oder andere Werte enthält.

In den Speicherzellen 243 und 244 finden Sie den Zeiger auf die aktuelle Position im Farb-RAM. Er wird immer dann vom Betriebssystem aktualisiert, wenn ein Zeichen ausgedruckt werden soll. Beim Ausdruck von Steuerzeichen (z.B. HOME) bleibt er dagegen auf der alten Position, da es zur Ausführung solcher Befehle nicht notwendig ist, auf das Color-RAM zuzugreifen. Beispiel: PRINT "(HOME)" läßt den Pointer unverändert, nach PRINT "(HOME)ABC" wird er dagegen auf die neue Cursorposition gesetzt.

Einen ähnlichen Zeiger gibt es auch für das Video-RAM. Allerdings ist er zweigeteilt. Die Register 209 und 210 bilden den Zeiger auf die Stelle im Speicher, an der die Zeile beginnt, in

der der Cursor gerade steht. Zu diesem Wert muß nur noch die aktuelle Spalte (0 - 39) aus Register 211 addiert werden, dann erhält man die Adresse des Bytes, das "unter" dem Cursor liegt.

Die Nummer der aktuellen Zeile (0 - 24) steht in Speicherzelle 214. Mittels der letzten beiden Register können wir den Cursor recht einfach auf dem Bildschirm positionieren. Die Spalte wird in Register 211 gePOKEd, die Zeile in 214 abgelegt. Das reicht allerdings noch nicht. Das Betriebssystem weiß dadurch noch nicht, daß der Cursor verschoben werden soll. Es gibt aber eine ROM-Routine, die diese Arbeit für uns übernimmt. Mit SYS 58732 kann sie aufgerufen werden. Demnach sieht die gesamte Befehlsfolge so aus:

```
POKE 211, Spalte: POKE 214, Zeile: SYS 58732
```

Dieser Trick wurde schon in Kap. 5.2. angewandt.

Haben Sie sich nicht auch schon gewünscht, den Cursor auch während einer Eingabe mit GET einschalten zu können? Das ist gar nicht schwierig! Speicherzelle 204 sagt dem Betriebssystem (genauer: der Interruptroutine), ob der Cursor erscheinen soll (in diesem Fall ergibt PEEK (204) 0), oder ob gerade wieder ein "Nickerchen" (natürlich nur für ein gewisses kleines Quadrat) angesagt ist. Starten wir ein Programm, so wird die Speicherzelle 204 vom Interpreter mit 1 geladen - der Cursor hört auf zu blinken.

Was das Betriebssystem kann, können wir schon lange. Mit POKE 204, 0 wird der Cursor einfach "während der Fahrt" eingeschaltet. Die Interruptroutine denkt nicht einmal daran, deswegen aufzumucken. Beim Ausschalten mit POKE 204, 1 müssen wir aufpassen, daß der Cursor nicht mitten in der Arbeit aufgehalten wird und ein reverses Zeichen unbeabsichtigt stehenbleibt. Auch hier gibt es aber Abhilfe. Register 207 gibt an, ob der Cursor gerade auf reverser oder normaler Darstellung (=0) ist. Die Zeilen

```
100 IF NOT (PEEK (207)) THEN 100
110 POKE 204,1
```

warten einfach solange, bis der Cursor gerade wieder einmal unsichtbar ist und schalten dann einfach ab.

Um bei der Eingabe zu bleiben, hier noch ein Tip für den nächsten INPUT-Befehl: Mit INPUT "text(crsr right)(crsr right)Z(crsr left)(crsr left)(crsr left)"; A$ wird das Zeichen Z bei Aufruf des INPUTs als Cursorzeichen benutzt. Sobald Sie eine Taste drücken, wird dieses Zeichen überschrieben.

Der nächste Befehl bezieht sich auf den Reverse-Modus. Unabhängig davon, ob der auszugebende String ein entsprechendes Steuerzeichen enthält oder nicht, wird mit POKE 199, 1 die umgekehrte Darstellung eingeschaltet. Abgeschaltet wird mit POKE 199,0.

Möchten Sie die Steuerzeichen eines Strings per Programm auf dem Bildschirm ausgeben? Bitte sehr - Speicherzelle 216 steht zur Verfügung. Sie gibt die Anzahl der noch ausstehenden Inserts an. Wie Sie wissen, werden Steuerzeichen im Insertmodus nicht ausgeführt, sondern lediglich als reverse Zeichen ausgegeben. Diesen Modus können Sie mit POKE 216, X einschalten, wobei X die maximale Anzahl der zu druckenden Zeichen darstellt.

Zu guter Letzt spielen wir noch einmal Betriebssystem. Wenn Sie schon einmal mit der Datasette gearbeitet haben, dann wissen Sie, daß während der Kassettenoperationen der Bildschirm ausgeschaltet wird. Auch hierfür ist wieder der VIC zuständig. In Speicherzelle 53265 sagt Bit 4, wie der Bildschirm aussehen soll. Mit POKE 53265, PEEK (53265) AND 239 wird ausgeschaltet, mit POKE 53265, PEEK (53265) OR 16 wieder eingeschaltet.

Zusammenfassung: Bildschirmtricks

Schriftfarbe ändern: POKE 646, Farbcode
Aktuelle Zeichenfarbe: PRINT PEEK (647)
Aktuelle Position im Color-RAM: PRINT PEEK (243) + 256 * PEEK (244)

Aktuelle Position im Video-RAM:

PRINT PEEK (209) + 256 * PEEK (210) + PEEK (211)

Cursorspalte: PRINT PEEK (211)
Cursorzeile: PRINT PEEK (214)

Cursor setzen:

POKE 211, Spalte: POKE 214, Zeile: SYS 58372

Cursor einschalten: POKE 204, 0
Cursor abschalten: POKE 207, 0: POKE 204, 1
INPUT mit speziellem Cursor: INPUT "text(2 x crsr right)Z(3 x crsr left)"; A$
Reverse-On: POKE 199,1
Reverse-Off: POKE 199,0
Sonderzeichen-Modus an: POKE 216, X
Bildschirm abschalten: POKE 53265, PEEK (53265) AND 239
Bildschirm einschalten: POKE 53265, PEEK (53265) OR 16

6. Hochauflösende Grafik

"Jetzt kommt Butter bei die Fische" (wie Tegtmeier das ausdrücken würde). Wir nähern uns nun den Gemächern der hochauflösenden Grafik, die leider von den Commodore-Ingenieuren recht überzeugend in den meterdicken Mauern des Betriebssystems versteckt wurden. Aber wie fast immer in diesem Buch ist es das gleiche Sesam-öffne-dich, das uns ans Ziel führt: ein POKE-Befehl.

6.1. Die Grafik-Modi

Wie auch bei der normalen Zeichendarstellung gibt es bei der hochauflösenden Grafik verschiedene Modi. Diesmal fehlt allerdings der Extended-Color-Mode (wozu sollte er auch bei Hochauflösung gut sein?). Im Normalmodus können wir 320 x 200 Punkte ansprechen. Diese 64000 Punkte lassen sich (ähnlich dem Charactergenerator) in 8000 Bytes unterbringen. Dieser achtmal größere Bildschirmspeicher heißt Bit-Map. Wie auf einer richtigen Landkarte geben die 1-Bits im Speicher an, ob in der Wirklichkeit (= Bildschirm) Hügel (= Punkte) vorhanden sind oder nicht. Die Farbe der Punkte gibt das Video-RAM an (wohlgemerkt: nicht das Color-RAM). Jedes Byte im ehemaligen Bildschirmspeicher ist dabei für einen Bereich zuständig, der im Normalmodus einem Zeichen entspricht. Die 4 höherwertigen Bits geben die Farbe (0 - 15) der Punkte an, die von 1-Bits repräsentiert werden, die 4 niederwertigen Bits die Farbe der Hintergrundpunkte (= 0-Bits).

Beim Multi-Color-Mode (jetzt aber hochauflösend!) wird die Punktematrix wieder eingeschränkt. Statt 320 x 200 Punkten haben wir jetzt nur noch 160 x 200 zur Verfügung. Da auch hier wieder je 2 Bits einen Punkt darstellen, brauchen wir 8000 Bytes für die Bit-Map, können damit aber auch 4 Farben pro Bildschirmzelle darstellen. Die Farben stammen nicht nur aus dem alten Video-RAM, sondern jetzt auch aus dem Hintergrundfarbregister 0 und dem Farb-RAM. Das Register 53281 (für die Hintergrundfarbe) wird für alle 00-Kombinationen benutzt. Bei

01 werden die höherwertigen, bei 10 die niederwertigen 4 Bits aus dem Video-RAM benutzt. Sind beide Bits auf 1, so holt sich der VIC den Farbcode aus dem entsprechenden Byte des Farb-RAMs.

6.2. Die Bit-Map

Zunächst etwas zur Lage der Bit-Map im Speicher. Dabei ist die Speicherzelle 53272 wieder von Bedeutung. Je nach Zustand des Bits 3 liegt die Bit-Map bei 8192 (wenn Bit 3 = 1) oder bei Speicherzelle 0. Letzteres nützt uns herzlich wenig, da sich dort die Zeropage befindet, die man ohne Wissen und Erlaubnis des Betriebssystems nicht überschreiben sollte.

Aufgebaut ist die Bit-Map wie ein Zeichengenerator (siehe Abb. 8). Die ersten 8 Bytes stellen die 8 Punktzeilen des ersten Quadrats (oder Zeichenblocks) dar usw. Deshalb kann es vorkommen, daß Sie nach dem Einschalten der Grafik normale Bildschirmzeichen auf dem Monitor sehen. Kopieren Sie einmal den Zeichengenerator in die Bit-Map und ändern Sie einige Bytes - das Ergebnis sollte Ihnen bekannt vorkommen.

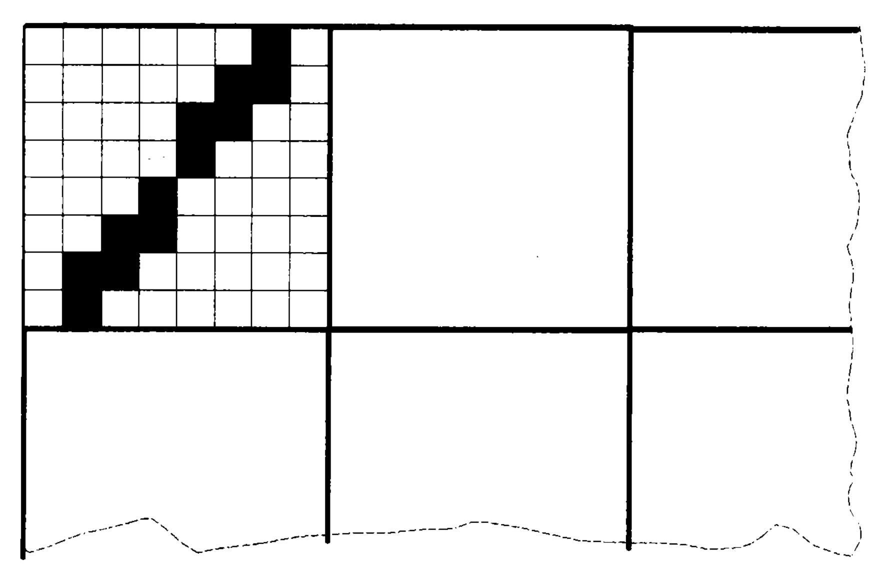


Abb. 8: Organisation der Bit-Map

Auch beim Multi-Color-Mode sieht es ähnlich aus. Nur sind hier jeweils 2 Bits für einen doppelt so breiten Punkt zuständig (aber das kennen Sie ebenfalls aus Kap. 5).

Die Lage der Bit-Map macht es unerläßlich, den BASIC-Speicher zu schützen. Da nur 8000 und nicht 8192 Bytes (was genau 8 K entspräche) benötigt werden, können wir den Speicher ab 16192 benutzen. Die Zeiger dafür sollten Sie eigentlich berechnen können (siehe Kap. 3.3.).

Da der normale BASIC-Speicher ab 2048 beginnt, haben wir sozusagen 6 Kilobytes übrig. Dieser freie Bereich könnte für Sprites eingesetzt werden. Weitere Ks sollten schließlich für die Farbgebung und weitere Bildschirmseiten reserviert werden. Wie Sie jetzt wissen, wird das Video-RAM als Farbspeicher mißbraucht. Dabei werden aber auch alte Texte überschrieben. Um dies zu vermeiden, sollte bei jedem Einschalten der hochauflösenden Grafik auf eine andere Bildschirmseite umgeschaltet werden (siehe Kap. 5.5.). Die beiden Modi beeinflussen sich somit nicht mehr gegenseitig. Einziger Nachteil: Die Sprite-Pointer

müssen jetzt zweimal gePOKEd werden, sofern die Sprites in beiden Modi benutzt werden: Einmal für den Zeichenmodus in 2040 bis 2047, das zweite Mal für die Hochauflösung in dem verschobenen Bereich. Leider funktioniert dies beim Multi-Color-Mode nicht ganz so elegant, da hier auch das Farb-RAM benutzt wird und sich dementsprechend die Farben des alten Textes ändern können.

6.3. Grafik einschalten

Um die Grafik einzuschalten, müssen wir drei Schritte vornehmen. Zunächst muß der Speicherbereich für die Bit-Map geschützt werden. Dies geschieht (wenn das Programm bereits fertig ist) durch ein Ladeprogramm (siehe Kap. 3.3.).

Bei der Programmerstellung sollten diese Befehle vorher im Direktmodus gegeben werden. Innerhalb des Programmes kann dann die Grafik eingeschaltet werden. Dazu muß Bit 5 in Speicherzelle 53265 auf 1 gesetzt werden. Damit weiß der VIC, daß keine Zeichen, sondern hochauflösende Grafiken dargestellt werden. Sollen mehrfarbige Grafiken eingesetzt werden, so muß zusätzlich noch das Multi-Color-Bit in Speicherzelle 53270 auf 1 gesetzt werden (genau wie im Zeichenmodus). Damit noch nicht genug. Die Lage der Bit-Map wird durch Bit 3 in Speicherzelle 53272 angezeigt. Daher muß auch dieses Bit auf 1 gesetzt werden. Schließlich sollten wir noch das Video-RAM verlegen, um den Bildschirminhalt nicht zu zerstören.

Wenn dies geschehen ist, sehen Sie ein ziemlich chaotisches Bild auf dem Monitor. Es fehlt noch der vierte Schritt! Mit einer FOR-NEXT-Schleife muß die Bit-Map gelöscht werden, ebenso das Video-RAM.

Hier die komplette Befehlsfolge:

```
POKE 43, 65: POKE 44, 63: POKE 16192, 0: NEW: REM
Speicher schützen (Direktmodus)
```

POKE 53265, 59: REM Grafik-Modus einschalten
(POKE 53270, 216: REM Multi-Color-Modus einschalten)
POKE 53272, 40: REM Bit-Map-Lage + Video-RAM-Verschiebung nach 2048

FOR I= 8192 TO 16191: POKE I, 0: NEXT: REM Bit-Map löschen

FOR I= 2048 TO 3047: POKE I, Punktfarbe * 16 + Hintergrundfarbe: NEXT: REM Farben setzen

Nach diesen POKEs finden wir das Video-RAM in den Speicherzellen von 2048 bis 3047 wieder. Ab 3072 stehen weitere 5 K für diverse Zwecke wie Sprites, Maschinenroutinen u.ä. zur Verfügung.

Da jede Grafik einmal ein Ende hat, hier die POKEs, die zum Ausschalten gebraucht werden:

POKE 53265, 155: REM Grafik-Modus ausschalten
(POKE 53270, 8: REM Multi-Color-Modus ausschalten)
POKE 53272, 21: REM Zeichensatz Großschrift einschalten

Haben Sie schon Ihre ersten Grafiken ausprobiert? Wenn ja, dann hat Sie sicher das langsame Löschen der Bit-Map geärgert. Deshalb folgt unten ein kleines Maschinenprogramm, das diese Aufgabe ungleich schneller bewältigt:

```
0 FOR I= 3600 TO 3659: READ A: POKE I,A: NEXT
1 DATA 169, 32, 133, 252, 169, 0, 133, 251, 162, 31, 160, 0, 145,
251, 136, 208, 251, 230, 252
2 DATA 202, 208, 246, 160, 64, 145, 251, 136, 16, 251, 169, 8,
133, 252, 165, 2, 162, 3, 160
3 DATA 0, 145, 251, 136, 208, 251, 230, 252, 202, 208, 246, 160,
232, 145, 251, 136, 208, 251
4 DATA 141, 0, 11, 96
```

Gestartet wird dieses Maschinenprogramm mit SYS 3600. Es löscht zunächst die Bit-Map, dann wird das Video-RAM (2048 - 3047) mit Punkt- und Hintergrundfarben geladen. Welche Farben dies sind, bestimmt Speicherzelle 2. Durch POKE 2, Punktfarbe * 16 + Hintergrundfarbe kann dies dem Programm mitgeteilt werden.

Die Maschinenroutine ist voll relokatibel, d.h. sie kann ebensogut im Kassettenpuffer oder andernorts stehen. Anfangsadresse ist immer das Byte, mit dem die FOR-NEXT-Schleife in Zeile 0 beginnt. Probieren Sie einmal die Geschwindigkeit aus. In Bruchteilen von Sekunden wird erledigt, was sonst einen längeren Zeitraum einnimmt.

Schließlich noch ein kleiner Tip. Wenn Sie Sprites, Grafikseite, Farben und Löschprogramm mit dem Hauptprogramm zusammen auf Diskette oder Cassette abspeichern wollen, so geht dies ziemlich einfach. Nach der Fertigstellung, wenn Sprites, Grafikseite und Maschinenprogramm schon im geschützten Bereich stehen, wird der Pointer in 43/44 auf den normalen BASIC-Anfang zurückgesetzt und dann ganz normal geSAVEd. Natürlich kann man, um Speicherkapazität zu sparen, den Pointer auf höhere Adressen zeigen lassen, wenn z.B. das Farb-RAM nicht mit abgespeichert werden soll. Wird das Programm dann (immer noch durch einen Lader, der die Zeiger setzt) mit LOAD "Name",8,1 zurückgeladen, so stehen Grafik, Sprites etc. bereits fix und fertig im Speicher. Dies ist besonders für Spielprogramme sehr nützlich.

Zusammenfassung: Grafik einschalten

Mit folgenden POKEs wird die hochauflösende Grafik bzw. Multi-Color-Grafik eingeschaltet. Das Video-RAM liegt in dieser Zeit zwischen 2048 und 3047, der BASIC-Start muß auf 16192 hochgesetzt werden.

```
POKE 53265, 59: REM Hochauflösung ein
(POKE 53270, 216: REM Multi-Color dazuschalten)
POKE 53272, 40: REM Bit-Map und Video-RAM verschieben
```

Danach müssen Video-RAM und BIt-Map noch gelöscht werden.

Ausgeschaltet wird so:
POKE 53265, 155: REM Grafik aus
(POKE 53270, 8: REM Multi-Color-Modus aus)
POKE 53272, 21: REM Großschrift einschalten

6.4. Punkte setzen

Wenn Sie einen bestimmten Punkt auf der Grafikseite setzen wollen, so können Sie auf Millimeterpapier zuerst das Bild aufzeichnen und dann die Kästchen in Byteinhalte umwandeln. Sollte sich jetzt bei Ihnen eine Vision aus Arbeit, Schweiß und Tobsuchtsanfällen aufbauen, so teilen Sie diese Erscheinung mit dem Autor dieses Werkes. Daher folgen jetzt zwei Routinen, die die Grafikprogrammierung erheblich erleichtern können.

6.4.1. Punkte setzen im Hochauflösungsmodus

Die unten aufgelistete Subroutine funktioniert vom Prinzip her genau wie die Blockgrafikroutine aus Kapitel 5.1. Da wir diesmal aber keine speziellen Grafikzeichen poken müssen, brauchen wir keine Tabelle, mit der die Punktkoordinaten umgesetzt werden können.

```
61000 REM Punkte setzen und löschen in Hochauflösung
61010 Y= 199 - Y: IF Y kleiner 0 OR Y größer 199 THEN RETURN
61020 IF X kleiner 0 OR X größer 319 THEN RETURN
61030 X1= INT (X/8): X2= INT (Y/8): AD= 8192 + 8*X1 + 320*X2 + (Y
AND 7)
61040 X3= 2 ^ (7- (X AND 7)): CA= 2048 + X1 + 40*X2
61050 POKE CA, (PEEK (CA) AND 15) OR 16 * CO
61060 IF L=1 THEN POKE AD, PEEK (AD) AND (255 - X3): RETURN
61070 POKE AD, PEEK (AD) OR X3: RETURN
```

Aufgerufen wird das Unterprogramm mit GOSUB 61000. Die Koordinaten X (0 - 319) und Y (0 - 199) geben den Ort des Punktes an. Der Koordinatenursprung liegt in der linken unteren Ecke. Sollten die Werte von X und Y nicht im zulässigen Bereich liegen, so wird die Routine in 61010 bzw. 61020 beendet. Damit ist es möglich, z.B. Linien scheinbar über den Bildschirmrand hinaus zu ziehen.

Zeile 61030 berechnet die Adresse des Bytes innerhalb der Bit-Map, das geändert werden soll. Dabei stellt INT (X/8) die Spalte dar, die im Farb-RAM belegt würde. Dieser Wert wird mit 8 multipliziert, da eine Zelle im Farb-RAM 8 Bytes in der Bit-Map repräsentiert. INT (Y/8) stellt analog dazu die Zeile im Farb-RAM dar. Um die Adresse der entsprechenden Zeile in der Bit-Map zu erhalten, wird dieser Wert mit der Anzahl der möglichen Punkte pro Zeile (320) multipliziert. Y AND 7 gibt schließlich die Zeile innerhalb des Farbquadrates an.

Die Variable X3 gibt den Wert an, der mit den schon gesetzten Bits verknüpft werden muß, um die gewünschten Punkte zu setzen oder löschen. AD enthält schließlich die POKE-Adresse für den Punkt, CA für die Farbe. Zeile 61050 POKEd die Farbe aus CO (0 - 15) in die 4 höherwertigen Bits der zugehörigen Farbspeicherzelle. Bitte beachten Sie, daß damit die Farbe für die Punkte des gesamten Quadrats geändert wird!

Ist die Variable L= 1, so bedeutet dies für die Routine, daß der angegebene Punkt gelöscht werden soll. In diesem Fall wird in Zeile 61060 verzweigt, andernfalls wird in der letzten Zeile der Funktionsteil "Setzen" aufgerufen.

Ein typischer Aufruf dieser Routine könnte so aussehen:

```
X= 100: Y= 25: REM Koordinaten setzen
CO= 2: L= 0: REM Farbe= ROT, Modus= SETZEN
GOSUB 61000: REM Routine aufrufen
```

6.4.2. Punkte im Multi-Color-Modus

Im Multi-Color-Modus müssen pro Punkt 2 Bits gesetzt werden, je nach Farbe verschiedene Kombinationen. Das legt die Methode nahe, pro Multi-Color-Punkt zweimal die Punktsetzroutine aufzurufen. Für das erste Bit wird einfach die X-Koordinate verdoppelt, für das zweite Bit wird die verdoppelte Koordinate noch um 1 erhöht. Da die zwei Bits aber auf jeden Fall im gleichen Byte liegen, kann auf den zweifachen Aufruf der Berechnung der POKE-Adresse verzichtet werden. Sehen Sie es sich selbst an:

```
61000 REM Multi-Color-Punkte setzen
61010 Y= 199 - Y: IF Y kleiner 0 OR Y größer 199 THEN RETURN
61020 X= 2*x: IF X kleiner 0 OR X größer 318 THEN RETURN
61030 x1= INT (X/8): X2= INT (Y/8): AD= 8192 + 8*X1 + 320*X2 + (Y
AND 7)
61040 X3= 2 ^ (7 - (X AND 7)): X4= 2 ^ (7 - ((X+1) AND 7))
61050 POKE AD, PEEK (AD) AND (255 - (X3 + x4))
61060 POKE AD, PEEK (AD) OR ((CO AND 1)*X3 + (CO AND
2)/2*X4):RETURN
```

Aufgerufen wird mit GOSUB 61000: Die Koordinaten sind wieder in X (0 - 159) und Y (0 - 199) abgelegt. Die Farben und der Setz- bzw. Löschmodus müssen nicht mehr angegeben werden, dafür die Farbkennzahl (0 -3) in CO, die die Bitkombination angibt. Soll ein Punkt gelöscht werden, so wird CO einfach auf 0 gesetzt. Die Farben, die durch die Bitkombination angesprochen werden sollen, müssen ggf. vorher mittels POKE in die entsprechenden Register geladen werden (für das Video-RAM übernimmt dies die Löschroutine). Bis auf eine kleine Änderung sind die Zeilen 61010 bis 61030 gegenüber dem normalen Hochauflösungsmodus nicht verändert.

Zeile 61040 berechnet die Verknüpfungsmasken für die beiden Bits (X3 und X4). Dann werden die beiden Bits zunächst gelöscht (Zeile 61050). Schließlich wird die Bitkombination durch

Zeile 61060 in das betreffende Byte eingeblendet. CO AND 1 gibt dabei das niederwertige Byte der Kombination an, (CO AND 2)/2 das höherwertige. Ist ein Bit 0, so wird das gesamte Produkt gleich 0. Folge: Das betreffende Bit aus dem Speicher wird mit 0 oder - verknüpft und bleibt demnach auf dem alten Stand, andernfalls wird mit 1 verknüpft - das Bit wird auf jeden Fall gleich 1. Fertig!

Hier noch ein Beispiel für einen typischen Aufruf:
X= 100: Y= 50: REM Koordinaten
CO= 2: REM Farbe aus höherwertigen Bits des Video-RAM holen
GOSUB 61000: REM Aufruf des Unterprogramms

6.5. Linien ziehen

Die folgende Unterroutine ist für beide Grafikmodi gleichermaßen geeignet. Sie benutzt die Punktsetzroutine aus dem Kap. 6.4. als Unterprogramm, daher beschränkt sie sich auf die Berechnung der Koordinaten für die einzelnen Punkte der Linie.

```
61100 REM Linien ziehen
61110 IF ABS (XE-XA) kleiner ABS (YE-YA) THEN 61160
61120 SP= (YE-YA)/ ABS(XE-XA+1E-20):YK= YA
61130 FOR XX= XA TO XE STEP SGN (XE-XA)
61140 YK= YK+SP: Y= INT (YK + .5): X= XX: GOSUB 61000
61150 NEXT XX: RETURN
61160 SP= (XE-XA)/ ABS(YE-YA+1e-20):XK= XA
61170 FOR XX= YA TO YE STEP SGN (YE-YA)
61180 XK= XK+SP: X= INT (XK + .5): Y=XX: GOSUB 61000
61190 NEXT XX: RETURN
```

Aufgerufen wird diesmal mit GOSUB 61100. Die Startkoordinaten der Linie werden in XA und YA, die Endkoordinaten in XE und YE übergeben. Je nach benutzter Punktsetzroutine müssen außerdem noch Farbe (CO) und Modus (Löschen/Setzen in L)

für Hochauflösung bzw. nur die Farbkennzahl für Multicolor angegeben werden.

Der Algorithmus ist eigentlich sehr simpel. Zunächst wird festgestellt, ob der Abstand der X-Koordinaten zueinander kleiner als der Abstand der Y-Koordinaten ist (Zeile 61110). Trifft dies zu, so wird der ganze Prozeß einfach umgedreht, die Funktionsweise bleibt aber gleich. Wozu das gut ist, zeigt sich bei den nächsten Schritten. Nehmen wir an, der X-Abstand ist größer als der Y-Abstand. Das heißt, daß mehrere Punkte der Linie die gleiche Y-Koordinate haben müssen, da wir auf dem Bildschirm nie eine wirklich schräg verlaufende Linie, sondern immer nur ein angenähertes Zick-Zack-Muster erzeugen können. Daher können wir einfach mit einer FOR-NEXT-Schleife (Zeile 61130) alle X-Koordinaten zwischen XA und XE "abklappern" und dazu die entsprechenden Y-Koordinaten berechnen. Wäre der X-Abstand kleiner als der Y-Abstand, so könnte es passieren, daß pro X-Koordinate mehrere Punkte in der Y-Richtung gesetzt werden müßten. Beim umgedrehten Verfahren wird daher einfach die Y-Richtung abgefahren und der X-Wert berechnet.

Diese Berechnung ist ebenfalls sehr simpel. Vor der Schleife wird die Schrittweite (SP) berechnet. Sie gibt den Abstand zweier aufeinanderfolgender Punkte in Y-Richtung an. Bei jedem Durchlauf der Schleife wird eine Hilfsvariable (YK) um diese Schrittweite erhöht. Dieser Wert wird gerundet (INT (YK + .5)) und dann als Y-Wert an die Punktsetzroutine übergeben (Zeile 61140).

Sollten die Koordinaten der zu setzenden Punkte nicht im erlaubten Bereich liegen, so wird dies von der Punktsetzroutine abgefangen. Leider ist dieses Unterprogramm nicht sehr schnell, doch für einfache Anwendungen (z.B. Funktionenplot) reicht es völlig aus. Legt man großen Wert auf Schnelligkeit, so sollte man diese Unterroutinen durch ein spezielles Hilfsprogramm wie die SUPERGRAPHIK 64 von DATA BECKER ersetzen. Sie ent-

halten sehr schnelle Befehle zum Arbeiten mit hochauflösender Grafik, Sprites usw.

6.6. Kreise zeichnen

Neben vielen anderen Figuren ist der Kreis eines der am häufigsten verwendeten grafischen Elemente. Er läßt sich auch nicht aus einem System von Linien aufbauen. Deshalb finden Sie unten eine entsprechende Subroutine. Sie ist zwar sehr langsam, doch meine ich, daß es besser ist, einen Kreis langsam zu zeichnen, als überhaupt nicht. Wie die Routine zum Linienziehen kann auch diese in beiden Grafikmodi verwendet werden.

```
61200 REM Kreise zeichnen
61210 FOR II= 0 TO 359
61220 XX = COS (II/180*PI) * R1
61230 YY = SIN (II/180*PI) * R2
61240 X = XX + MX: Y = YY + MY
61250 GOSUB 61000
61260 NEXT II: RETURN
```

Der Aufrufbefehl ist GOSUB 61200. Die Übergabevariablen sind MX und MY für die Koordinaten des Kreismittelpunktes sowie R1 und R2 für die Radien (die in den Anzahlen der Punkte angegeben werden, die die Radien messen sollen) und die bekannten CO und L für "normale" Hochauflösung bzw. CO als Farbkennzahl für Multi-Color.

Sie werden sich jetzt fragen, wozu zwei verschiedene Radien gut sein sollen. Ganz einfach. Wenn Sie R1 und R2 verschieden wählen, erhalten Sie eine Ellipse!

Wenn Sie sich das Entstehen eines Kreises auf dem Bildschirm ansehen, dann ahnen Sie vielleicht schon das Funktionsprinzip. Grundlage sind die trigonometrischen Funktionen SINUS und

COSINUS, mit denen sich Koordinaten auf dem Einheitskreis berechnen lassen.

In einer Schleife, die den gesamten Kreisumfang (0 bis 360 Grad) durchläuft, werden zu jedem Punkt auf dem Rand zwei Koordinaten (XX und YY) berechnet, die die Entfernung zum Mittelpunkt angeben. Addiert man diese Werte zu MX und MY, so erhält man die konkreten Bildschirmkoordinaten, die an die Punktsetzroutine übergeben werden.

Hier noch ein Beispielaufruf:

```
MX = 120: MZ = 100
R1 = 50: R2 = 30
CO = 1: L = 0
GOSUB 61200
```

Viel Spaß beim "Kreisen"!

7. Sprites

Die Sprites stellen das bekannteste Ausstattungsmerkmal des 64ers dar. Keine andere Grafikart ist so vielseitig einzusetzen. Das ist vielleicht auch der Grund, warum von allen Grafikmöglichkeiten nur diese im CBM-Handbuch beschrieben ist. Doch auch hier hat Commodore durch einen unerfindlichen Ratschluß wieder heillose Verwirrung gestiftet. Wo ist die Kontrolle von Spritekollisionen erklärt? Wie erzeuge ich Multi-Color-Sprites?

Die verschiedenen Möglichkeiten, die auch die Sprites bieten, sind in den folgenden Abschnitten beschrieben. Damit können Sie Commodore ein Schnippchen schlagen!

7.1. Multi-Color-Sprites

Ja, Sie haben eben richtig gelesen. Neben den normalen, hochauflösenden "Mini-Grafiken" läßt sich der VIC auch auf Multi-Color-Sprites programmieren. Das ist gar nicht schwer.

Eingeschaltet wird der Multi-Color-Modus durch das Setzen des der Sprite-Nummer entsprechenden Bits im VIC-Register 28 (53276). Um z.B. Sprite 6 im Multi-Color-Mode zu definieren, benutzt man diese Folge:

```
POKE 53276, PEEK (53276) OR (2^6)
```

Natürlich kann durch

POKE 53276, PEEK (53276) AND (255-2^6) das Bit wieder auf 0 gesetzt werden.

Damit wäre das Sprite schon auf Mehrfarbenbetrieb umgeschaltet. Da wieder je 2 Bits der Matrix einen Punkt darstellen, bleiben uns nur noch 12 x 21 Punkte. Wie dieser Modus funktioniert, wissen Sie , es fehlt nur noch die Information, welche Bitkombinationen welche Farben erzeugen.

Sind beide Bits auf 0, so ist der betreffende Punkt des Sprites transparent, d.h. der Hintergrund (z.B. ein Buchstabe) scheint an dieser Stelle durch das Sprite hindurch. Ist das niederwertige der beiden Bits auf 1, so holt sich der VIC die Farbe aus den Multi-Color-Registern 37 und 38 (53285 und 53286). Welches der beiden benutzt wird, entscheidet Bit 2. Ist es 0, so wird Register 37 benutzt, sonst 38. Bei der Kombination 1 0 stammt die Farbinformation aus dem normalen Farbregister des Sprites. Diese Farbe kann für jedes Sprite verschieden sein, nicht jedoch die Multi-Color-Farben. Diese stammen für alle Sprites aus den gleichen Registern und dürfen nur im Bereich von 0 bis 7 liegen. Multi-Color-Sprites werden genauso definiert wie normale, lediglich die Zuordnung von Bits zu Punkten ist anders. Auch die Koordinaten auf dem Bildschirm bleiben gleich. Der größte Vorteil ist wohl, daß man verschiedene Sprite- und Grafikmodi mischen kann. So können Hochauflösungs- und Multi-Color-Sprites nebeneinander auf dem Bildschirm stehen. Überdies ist es dem VIC egal, ob auf dem Bildschirm gerade Zeichen oder Grafiken stehen.

Einzige Einschränkung: Während eines Floppyzugriffs sollte man darauf achten, daß die Sprites ausgeschaltet sind (POKE 53269, 0), da der VIC den Taktablauf regelt und dieser um so mehr Zeit braucht, je mehr Sprites auf dem Bildschirm sichtbar sind. Das kann die Datenübertragung stören.

Zusammenfassung: Multi-Color-Sprites

Modus einschalten: entsprechendes Bit in Reg. 28 (53276) auf 1 setzen.

Farben aus Reg. 37 (bei 0 1) und aus Reg. 38 (bei 1 1) sowie aus normalem Farbenregister für das Sprite (bei 1 0).

Sprites und Grafiken der verschiedenen Modi können gemischt werden.

7.2. Kollisionen

Der VIC zeigt jede Berührung eines Sprites mit einem anderen Sprite oder Bildschirmpunkt in seinen Registern an. Erfolgt eine Kollision zwischen zwei oder mehreren Sprites, so erscheint dies in Register 30 (53278). Die Nummern der beteiligten Sprites werden durch das Setzen der entsprechenden Bits in diesem Register angezeigt. Mit

```
PRINT PEEK (53278) AND 2^n
```

können Sie demnach feststellen, ob das Sprite n an der Kollision beteiligt war. Ist dies der Fall, so liefert der obige Befehl die Zahl n als Ergebnis die Zahl 2^n als Ergebnis, sonst 0. Die Kollision wird nur dann angezeigt, wenn sich wirklich 2 Punkte berühren, nicht aber, wenn sich zwei Sprites in Bereichen überlagern, die völlig punktleer sind. Die Bits bleiben solange gesetzt, bis Sie sie auslesen POKE (d.h. jeder PEEK(!)-Befehl löscht das Kollisionsregister). Es kann also vorkommen, daß eine Berührung angezeigt wird, obwohl sich die Sprites längst wieder voneinander entfernt haben. Ich empfehle daher, vor jedem Abfragen dieser Register alle Bits zu löschen. Besteht noch eine Kollision, so werden die entsprechenden Zahlen sofort wieder gesetzt, so daß der dann folgende PEEK-Befehl dies entdecken wird.

Die Kontrolle von Sprite-Hintergrund-Kollisionen erfolgt in gleicher Weise. Die entsprechenden Bits werden auf 1 gebracht, wenn in der Bit-Map oder im Zeichengenerator ein vom Sprite überlagerter Punkt durch eine 1 repräsentiert wird. Mit anderen Worten: Jede Berührung eines Sprites mit einem Zeichen oder Grafikpunkt wird registriert. Zuständig ist diesmal Register 31 (53279).

Zur Veranschaulichung der Programmierung von Sprite-Kollisions-Kontrollen ist unten ein kleines Spiel aufgelistet. Dabei handelt es sich um ein sehr simples Autorennen. Ziel ist es, mittels der Tasten Z (= LINKS) und / (= RECHTS) einem Hindernisauto auszuweichen. Jede Berührung des eigenen Wagens mit dem Fahrbahnrand oder dem anderen Wagen wird in den

Kollisionsregistern registriert und führt zu einem CRASH. Selbstverständlich können die REMs beim Eintippen weggelassen werden; sie würden das Spiel ohnehin nur verzögern.

```
1 REM +++++++++++++++++++++++++++++++++++
2 REM A U T O - R E N N E N  VERSION C-64
3 REM +++++++++++++++++++++++++++++++++++
10 PRINT CHR$(147):POKE53280,0:POKE53281,0:V=53248:REM BILDSCHIRM
VORBEREITEN
20 FORI=832TO894:READA:POKEI,A:NEXT:REM AUTO-SPRITE EINLESEN
30 FORI=896TO958:READA:POKEI,A:NEXT:REM CRASH-SPRITE EINLESEN
40 POKE2040,13:POKE2041,13:POKEV+39,1:POKEV+40,2:REM SPRITEPOINTER
& FARBEN
45 POKEV+23,3:POKEV+29,3:REM SPRITES VERGROESSERN
50 FORI=0TO24:POKE1034+I*40,160:POKE55306+I*40,1
60 POKE1055+I*40,160:POKE55327+I*40,1:NEXTI:REM FAHRBAHN AUSGEBEN
70 POKEV,168:POKEV+1,170:POKEV+21,3:X=168:REM STARTPOS. AUTO &
SPRITES EIN
80 POKEV+2,168:POKEV+3,0:HX=168:HY=0:REM STARTPOSITION HINDERNIS
90 POKEV+30,0:POKEV+31,0:REM KOLLISIONSKONTROLLE LOESCHEN
100 A=PEEK(203):REM TASTATURABFRAGE
110 IFA=12THENX=X-1:REM Z GEDRUECKT
120 IFA=55THENX=X+1:REM / GEDRUECKT
130 POKEV,X:REM AUTO BEWEGEN
140 IFPEEK(V+30)<>0ORPEEK(V+31)=1THENPOKE2040,14:FORI=0TO500:NEXT:
RUN:REM CRASH
150 HY=HY+2:IFHY>240THENHY=30:REM BEWEGUNG NACH UNTEN
160 HX=HX+INT(RND(TI)*5)-2:IFHX<120THENHX=120:REM KOOR. & LINKER
RAND
170 IFHX>216THENHX=216:REM RECHTER RAND?
180 POKEV+2,HX:POKEV+3,HY:GOTO100:REM HINDERNIS BEWEGEN
1000 REM SPRITE-DATA AUTO
1100 DATA 0,0,0
1101 DATA 0,126,0
1102 DATA 0,126,0
1103 DATA 0,255,0
1104 DATA 12,255,48
1105 DATA 15,255,240
1106 DATA 12,255,48
```

```
1107 DATA 0,255,0
1108 DATA 0,255,0
1109 DATA 1,231,128
1110 DATA 1,195,128
1111 DATA 1,195,128
1112 DATA 1,195,128
1113 DATA 3,195,192
1114 DATA 3,195,192
1115 DATA 115,255,206
1116 DATA 115,255,206
1117 DATA 127,255,254
1118 DATA 115,255,206
1119 DATA 115,255,206
1120 DATA 0,0,0
2000 REM SPRITE-DATA CRASH
2100 DATA 123,20,0
2101 DATA 0,24,0
2102 DATA 30,44,77
2103 DATA 21,126,3
2104 DATA 240,125,48
2105 DATA 15,205,240
2106 DATA 22,155,41
2107 DATA 1,205,156
2108 DATA 0,155,0
2109 DATA 1,201,108
2110 DATA 1,105,108
2111 DATA 1,095,028
2112 DATA 1,155,158
2113 DATA 23,115,192
2114 DATA 32,242,132
2115 DATA 35,239,216
2116 DATA 1,095,028
2117 DATA 32,242,132
2118 DATA 68,155,0
2119 DATA 27,125,48
2120 DATA 0,0,0
```

Zusammenfassung: Kollisionen

Berührungen von Sprites mit anderen Sprites oder Hintergrundzeichen werden durch Setzen der entsprechenden Bits angezeigt. Dies geschieht in den Registern 30 (53278) und 31 (53271) des VIC. Die Bits bleiben solange gesetzt, bis der Anwender sie löscht.

7.3. Prioritäten & Bewegungsbereich

Wußten Sie, daß es verschiedene Möglichkeiten der Überlagerung von Bildschirmzeichen, Grafik und Sprites gibt? Im Normalfall stehen die Sprites vor dem aktuellen Bildschirminhalt. Doch oft ist es wünschenswert, daß die Zeichen vor dem Sprite stehen (z.B. wenn ein Flugzeug hinter einem Haus herfliegen soll). Auch hierfür hat der VIC wieder ein Register "in Reserve".

Das Register hat die Adresse 53275 (V+27). Wird hier ein Bit auf 1 gesetzt, so bedeutet dies, daß das zugehörige Sprite hinter den Bildschirmzeichen abgebildet wird. Im Normalfall sind alle diese Bits auf 0, das Sprite hat also gegenüber den Zeichen höhere Priorität.

Kurios wird es, wenn mehrere Sprites mit verschiedenen Prioritäten abgebildet werden. Wie Sie wissen, wird das Sprite mit der kleinsten Nummer immer vor seinen Kollegen abgebildet. Ist das vorderste Sprite aber auf niedrige Priorität gegenüber den Bildschirmzeichen eingestellt, so erscheint es zwar unter der Schrift, aber immer noch vor den anderen Sprites, auch wenn diese Vorrang vor den Zeichen hätten. Damit lassen sich leicht optische Täuschungen erzeugen.

Haben Sie schon einmal Grafiken erzeugt und versucht, Sprites damit in Deckung zu bringen? Wenn ja, dann werden Sie festgestellt haben, daß die Koordinaten von Sprites und Grafik nicht übereinstimmen. Der Bewegungsbereich der Sprites wurde größer definiert, um ein "Herausfahren" aus dem Bildschirm zu ermöglichen. In die linke obere Ecke können Sie eine solche "Grafik in der Grafik" mit den Koordinaten 24 und 50 bewegen.

Diese beiden Zahlen stellen die Korrekturfaktoren dar, die man zu den Grafik-Koordinaten addieren muß, um das Sprite richtig zu positionieren. Die Mitte des Bildschirms erreicht man also mit den Werten 160+24 und 100+50 (alles auf die linke obere Ecke des Sprites bezogen).

Zusammenfassung: Prioritäten und Bewegungsbereich

Sprite-Priorität (vor/hinter Zeichen) regelt das zugehörige Bit in VIC-Register 27 (53275). Durch Setzen des Bits wird das Sprite hinter den Zeichen abgebildet.

Korrekturfaktoren für Sprites gegenüber Grafik-Koordinaten:

24 (X-Richtung) und 50 (Y-Richtung)

7.4. Ideen für die Sprite-Programmierung

Viele Spielprogramme benutzen die sogenannte Animation, um z.B. einen kleinen Sprite-Mann möglichst naturgetreu zu seiner Sprite-Frau laufen zu lassen. Das sieht dann so aus, als würden Arme und Beine einzeln bewegt. Auf den zweiten Blick stellt man allerdings fest, daß es im Grunde nur zwei oder drei verschiedene Positionen für Arme und Beine gibt. Das Prinzip dieser Animation ist damit klar. Ein Sprite besteht in diesem Fall aus zwei getrennten Blöcken, zwischen denen während der Fortbewegung immer wieder umgeschaltet wird. In einem Block ist das Sprite mit geschlossenen Armen und Beinen definiert, im anderen schreitet es gerade weit aus. Werden diese beiden Bilder mittels der Pointer (2040 - 2047) abwechselnd eingeschaltet, so entsteht der Eindruck einer laufenden Figur. In Wirklichkeit werden nur die "Vorlagen" für die Sprites ständig ausgewechselt. So einfach ist das. Voraussetzung ist natürlich, daß genug Speicherplatz für die verschiedenen Bilder vorhanden ist. Hier sollte man wieder den BASIC-Anfang in weiter oben liegende Bereiche verlegen. Wenn Sie hochauflösende Grafik benutzen, haben Sie ja sowieso genug Platz.

Oft ist es auch nützlich, wenn man die Sprite-Blöcke auf Diskette oder Cassette abspeichert. Ein Programm dazu kennen Sie bereits aus Kapitel 4.1.

Interessant erscheint mir auch die Idee, die hochauflösenden Sprites als kleine Grafikbildschirme innerhalb des Zeichenmodus zu "mißbrauchen". Nehmen wir an, Sie möchten den Graphen einer beliebigen Funktion in Hochauflösung darstellen, gleichzeitig aber auch ein paar Kommentare dazusetzen. Eine Möglichkeit ist, die Punktmatrizen der benötigten Buchstaben aus dem Zeichengenerator auszulesen und durch entsprechendes Setzen von Grafikpunkten die Zeichen künstlich zu erzeugen. Aber das ist sehr umständlich, ebenso wie der umgekehrte Weg, bei dem der Zeichensatz so umdefiniert wird, daß die für den Graphen nötigen Punkte innerhalb der Zeichenmatrix gesetzt werden. Es bleiben die Sprites. Man nehme derer 4, ordne sie im Quadrat auf dem Bildschirm an der gewünschten Position an und berechne für jeden Punkt die zu setzenden Bits innerhalb der Spritematrix. Dies erledigt für uns die untenstehende Routine. Ich verzichte diesmal auf eine nähere Erklärung, da das Programm vom Aufbau und der Funktionsweise her der Punktsetzroutine für hochauflösende Grafik entspricht.

```
10 FOR I= 704 TO 767: POKE I, 0: NEXT I: REM Sprite 11 löschen
20 FOR I= 832 TO 1023: POKE I, 0: NEXT I: REM Sprites 13-15 lö-
schen
30 V= 53248: POKE V,100: POKE V+1, 100: POKE V+2, 148: POKE V+3,
100: REM Position Sprites 0 und 1
40 POKE V+4, 100: POKE V+5, 142: POKE V+6, 148: POKE V+7, 142: REM
Position Sprites 2 und 3
50 FOR I= 39 TO 42: POKE V+I, 1: NEXT I: REM Farben setzen
60 POKE V+23, 15: POKE V+29, 15: POKE V+21, 15: REM Vergrößern und
einschalten
100 PRINT CHR$(19);:INPUT"X-KOOR";X:INPUT"Y-KOOR";Y
110 GOSUB 62000:GOTO 100
62000 Y= 41-Y: IF Y kleiner 0 OR Y größer 42 THEN RETURN
62010 IF X kleiner 0 OR X größer 47 THEN RETURN
62020 BX= INT (X/24): BY= INT (Y/21): IF BX=0 AND BY=0 THEN
BA=704: GOTO 62040
```

```
62030 BA= 768 + BX*64 + BY*128
62040 BX= X - 24*BX: BY= Y - 21*BY
62050 X1= INT (BX/8): X2= 7- (BX AND 7): X3= BY*3
62060 AD= BA + X3 + X1
62070 IF L=1 THEN POKE AD, PEEK (AD) AND (255- 2^X2): RETURN
62070 POKE AD, PEEK (AD) OR (2^X2)
```

Aufgerufen wird mit GOSUB 62000. Wie bei der hochauflösenden Grafik werden die Koordinaten in X und Y, die Modusangabe (Setzen/Löschen) in L übergeben. Es werden die Sprites 0 bis 3 und die Blöcke 11, 13, 14, 15 benutzt. In der vorgestellten Version wurden die Sprites in beide Richtungen vergrößert. Wer möchte, kann sie auch in normaler Größe erscheinen lassen, muß dann aber die Positionen (siehe Zeilen 30 und 40) korrigieren.

Innerhalb der 4 Sprites können 48 x 42 Punkte gesetzt werden. Da es sich nicht um Multi-Color-Sprites handelt, haben alle Punkte die gleiche Farbe. Diese wird in Zeile 50 festgelegt.

Wer möchte, kann sich die Routine zum Linienzeichnen auf Sprite-Graphik umschreiben, es ist gar nicht schwer. Alle Sonderfunktionen (wie Priorität, Kollisionen etc.) können wie üblich auch für die Sprites 0 - 3 eingesetzt werden, da das Unterprogramm nur auf die Punktmatrizen in den Bereichen 704 - 766, 832 - 894, 896 - 958 und 960 - 1022 wirkt. Es lohnt sich also, ein wenig zu experimentieren.

Damit ist das Kapitel über die Programmierung mit Sprites zu Ende. Dies sollte Sie jedoch auf keinen Fall davon abhalten, weiter mit den VIC-Registern zu experimentieren. Es gibt viele Einsatzmöglichkeiten für die Sprites, die noch der Entdeckung harren!

8. Tonerzeugung

Was der VIC für den Bildschirm ist, stellt der SID (Sound Interface Device) für die Töne dar. Dieser bietet für jede Möglichkeit der Tonerzeugung ein entsprechendes Register. Leider wurden auch diese nicht ausführlich im CBM-Handbuch beschrieben. Da aber die Darstellung aller Möglichkeiten des SID viel zu umfangreich wäre, folgen hier nur die Grundtechniken der Soundprogrammierung.

8.1. Die Arbeitsweise des SID

In diesem Abschnitt soll im Vordergrund stehen, was im Computer abläuft, wenn ein Ton erzeugt werden soll.

Wird ein bestimmtes Startbit auf 1 gesetzt, so sieht der SID zuerst nach, welche Frequenz der Ton haben soll. Dann erzeugt er eine entsprechende Schwingung. Diese wird durch eine Art "elektronische Töpferscheibe", den Wellenformmodulator geschickt. Dadurch erhält der Ton die einprogrammierte Wellenform (Dreieck, Rechteck, Sägezahn, Rauschen) und deren charakteristisches Klangbild.

Dann formt der SID den Tonverlauf anhand der sogenannten Hüllkurve. Sie gibt an, welche Lautstärke der Ton in den verschiedenen Phasen hat. Die Hüllkurve setzt sich dazu aus 4 Parametern zusammen. Der Anschlag bestimmt, wie schnell die in einem eigenen Register angegebene Höchstlautstärke erreicht wird. Danach schwillt der Ton bis zu einem bestimmten Wert wieder ab, der im Parameter "HALTEN" zu finden ist. Die Geschwindigkeit dieses Abschwellens zeigt der Parameter "Abschwellen" an. Die jetzt erreichte Laustärke bleibt erhalten, bis das Startbit wieder auf 0 gesetzt wird. Der Parameter "AUSKLINGEN" gibt die Geschwindigkeit vor, mit der der Ton abgeschaltet wird. Damit kann ein Nachhall erzeugt werden.

Bei Rechteckschwingungen kann zusätzlich noch das sogenannte Tastverhältnis abgegeben werden, das das Verhältnis zwischen

Impuls-Ein und Impuls-Aus regelt. Auch damit kann die Klangfarbe beeinflußt werden.

Weitere Möglichkeiten (die hier aber nicht beschrieben werden sollen) sind z.B. Ringmodulation, bei der der Ton einer Stimme in Abhängigkeit der beiden anderen erzeugt wird, und Filter, die verschiedene Frequenzbereich ausfiltern können.

8.2. Die Programmierung

Jetzt geht es zur Sache. In diesem Kapitel werden wir die Programmierung von Tönen und Tonfolgen beschreiben. Dabei werden wir zum Teil von der im CBM-Handbuch vorgestellten Methode abweichen, da diese die Nutzung eines Teils der Möglichkeiten schlicht und einfach unmöglich macht.

Egal wie Ihr Soundprogramm aussieht, eines sollte immer ganz am Anfang stehen: Die Lautstärke. Sie wird in die 4 niederwertigen Bits des Registers 24 (54296) gePOKEd. Ein Versuch, dieses oder eines der anderen Register (von 0 bis 24) mittels PEEK auszulesen, wird immer zum Scheitern verurteilt sein. Aufgrund einer besonderen Konstruktion lassen sich diese Bytes nicht auslesen, sondern nur beschreiben. Ein PEEK-Befehl kann daher unsinnige Ergebnisse liefern.

Genau umgekehrt verhält es sich mit den Registern 25 - 28. Hier kann nur gelesen werden, ein POKE bleibt dagegen unwirksam.

Doch zurück zur Musik. Da die 4 höherwertigen Bits des Lautstärkeregisters im Normalfall auf 0 sein sollten, können wir die gewünschte Zahl einfach einPOKEn. Mit POKE 54296,0 wird demnach die Lautstärke ganz zurückgenommen, mit POKE 54296,15 dagegen können wir die volle Lautstärke erzeugen. Die Lautstärke kann immer nur für alle drei Stimmen gleichzeitig gesetzt werden.

Als nächstes folgt die Frequenz, also die Tonhöhe. Sie können zwischen 65536 verschiedenen Frequenzen wählen. Welche Sie

nehmen, bleibt Ihnen überlassen. Beim Programmieren von Melodien ist die Notentabelle im Anhang des Handbuches nützlich. Wie Sie die Frequenzzahl in High- und Low-Byte aufspalten können, wissen Sie aus dem Kapitel über Zeiger. Diese Zahlen werden in die Register 0 und 1 (für Stimme 1), 7 und 8 (für Stimme 2) oder 14 und 15 (für Stimme 3) gePOKEd.

Jetzt sollten Sie daran gehen, die Hüllkurve festzulegen. Für den Anschlag und die Dauer des Abschwellens ist Register 5 (bzw. 12 oder 19) zuständig. Der Anschlag ist in den höherwertigen Bits zu Hause, der Wert für das Abschwellen in den niederwertigen. Ähnlich geht es den Werten für Halten und Ausklingen in Register 6, 13 oder 20., wobei "Halten" die höherwertigen Bits belegt. Ist dieser Wert 0, so bleibt die Stimme stumm. Ansonsten stellt er die Lautstärke des Tons im Verhältnis zur Höchstlautstärke aus Register 24 dar.

Wollen Sie Rechteckschwingungen benutzen, so braucht der SID noch das Tastverhältnis. Es kann Werte zwischen 0 und 4095 annehmen und liegt in den Registerpaaren 2/3, 9/10 oder 16/17. Von den höherwertigen Bytes dieser Paare werden jeweils nur die ersten (niederwertigen) Bits benutzt. Höhere Zahlen als 15 in einem dieser Register machen also keinen Unterschied.

So weit - so gut. Bisher sind wir wie im Handbuch vorgegangen. Um die Wellenform festzulegen, sollten wir Register 4 (bzw. 11 oder 18) betrachten. Ähnlich einigen VIC-Speicherzellen hat auch hier jedes Bit eine eigene Bedeutung. Bit 0 stellt das schon erwähnte Start-Stop-Bit für den Tonablauf dar. Wird es auf 1 gesetzt, so wird der Ton der zugehörigen Stimme eingeschaltet und der Hüllkurvenablauf gestartet. Wird es wieder auf 0 gesetzt, so wird der Ton je nach Hüllkurve in einer entsprechenden Zeit beendet. Bitte beachten Sie bei der Programmierung, daß der VIC in der Ausklingzeit möglichst keinen neuen Ton mit der gleichen Stimme erzeugt. Sollen z.B. für Melodien schnell aufeinanderfolgende Töne programmiert werden, so empfiehlt es sich, für die Ausklingzeit einen sehr kleinen Wert zu wählen.

Die Bits 1 und 2 des Registers 4 dienen Steuerungszwecken. Bit 3 ist für uns wieder sehr nützlich. Sollten zwei oder mehr Wellenformen gleichzeitig eingeschaltet worden sein, so kann der SID blockieren, d.h. es wird kein Ton mehr erzeugt. Durch Setzen des Bits 3 und durch Löschen der Wellenformen kann dies aufgehoben werden. Mittels POKE 54276, 8 wird der SID also neu initialisiert.

Die Bits 4 bis 7 bestimmen schließlich die Wellenform. Ist ein Bit auf 1 gesetzt, so wird die entsprechende Form erzeugt, wobei Bit 4 für Dreiecks-, Bit 5 für Sägezahn- und Bit 6 für Rechteckschwingungen zuständig ist. Bit 7 erzeugt ein Rauschen. Diese Schwingungsformen können auch gemischt werden.

Um einen Ton einzuschalten, müssen Wellenform und Start-Bit gleichzeitig gePOKEd werden. Daraus ergeben sich die im CBM-Handbuch angegebenen Codes für die verschiedenen Klänge (17, 33, 65, 129). Zum Ausschalten darf jedoch nicht einfach Reg. 4 (bzw. 11 oder 18) mit 0 geladen werden. Das käme dem Abwürgen eines Motors mitten auf der Autobahn gleich. Der SID findet plötzlich keine Angabe über die Wellenform und kann so auch keine Hüllkurve ordnungsgemäß beenden (womit auch?). Das Ergebnis ist der typische Ausschaltknack. Wird dagegen nur Bit 0 gelöscht, so entfällt das Knacken und der Ton klingt weich aus. Dies erreicht man durch POKEn der Formcodes -1 (also 16, 32, 64 oder 128). Und siehe da, wir haben unsere Bits als Zweierpotenzen zurück.

Zur Verdeutlichung der Vorgehensweise und als Ersatz für das nicht lauffähige Listing aus dem 64er-Handbuch hier ein Programm, das das Spielen von Melodien per Tastatur ermöglicht:

```
10 PRINT CHR$(147)
20 PRINT " W E   T Y U"
30 PRINT " A S D F G H J K"
100 S= 54272
110 POKE S+24, 15: REM Lautstärke
120 POKE S+5, 136: REM Anschlag & Abschwellen
130 POKE S+6, 248: REM Halten & Ausklingen
```

```
140 POKE S+4, 8: REM SID initialisieren
150 FOR I= 0 TO 40: NEXT I: POKE S+4, 16: REM Startbit=0
160 GET A$: IF A$="" THEN 160
170 IF A$= "A" THEN POKE S, 207: POKE S+1, 34
180 IF A$= "S" THEN POKE S,  18: POKE S+1, 39
190 IF A$= "D" THEN POKE S, 219: POKE S+1, 43
200 IF A$= "F" THEN POKE S, 118: POKE S+1, 46
210 IF A$= "G" THEN POKE S,  39: POKE S+1, 52
220 IF A$= "H" THEN POKE S, 138: POKE S+1, 58
230 IF A$= "J" THEN POKE S, 181: POKE S+1, 65
240 IF A$= "K" THEN POKE S, 157: POKE S+1, 69
250 IF A$= "W" THEN POKE S, 225: POKE S+1, 36
260 IF A$= "E" THEN POKE S, 101: POKE S+1, 41
270 IF A$= "T" THEN POKE S,  58: POKE S+1, 49
280 IF A$= "Y" THEN POKE S,  65: POKE S+1, 55
290 IF A$= "U" THEN POKE S,   5: POKE S+1, 62
300 POKE S+4, 17: GOTO 150
```

Die Zeilen 10 - 30 verdeutlichen die Tastaturbelegung. Dann folgt der Vorbereitungsteil (100 - 140). In Zeile 150 befindet sich eine kleine Warteschleife, die das ordnungsgemäße Ausklingen des Tons ermöglicht. Außerdem wird durch den POKE-Befehl das Start-Stop-Bit auf 0 gesetzt. Wer die Wellenform wechseln möchte, kann die Werte in dieser und in Zeile 300 ändern. Die Funktion der Zeilen 160 - 290 dürfte klar sein; hier wird die Frequenz entsprechend der gedrückten Taste gesetzt.

Wer nähere Informationen über die Tonerzeugung haben möchte, sollte sich Spezialliteratur, z.B. das Musikbuch von DATA BECKER besorgen. Hier werden auch ausgefallene Programmiertechniken beschrieben.

Zusammenfassung: Tonerzeugung

Lautstärke in Reg. 24. Bereich: 0 - 15.
Anschlag: höherwertiges Halbbyte in Reg. 5/12/19
Abschwellen: niederwertiges Halbbyte in Reg. 5/12/19
Halten: höherwertiges Halbbyte in Reg. 6/13/20

Ausklingen: niederwertiges Halbbyte in Reg. 6/13/20
Wellenform: Register 4/11/18
Bit 4: Dreieck
Bit 5: Sägezahn
Bit 6: Rechteck
Bit 7: Rauschen
außerdem: Bit 3: Initialisierung
Bit 1: Start-Stop-Bit
Frequenz: Registerpaare 0/1, 7/8 oder 14/15

9. Die Tastatur

Schon rein äußerlich stellt die Tastatur das auffälligste Merkmal des 64ers dar. Kaum ein Computer dieser Preisklasse ist mit einer qualitativ so hochwertigen Tastatur ausgestattet. Daß sie nicht nur Schreibkomfort, sondern auch Programmiertricks ermöglicht, soll Ihnen dieses Kapitel zeigen.

9.1. Aufbau und Funktionsweise der Tastatur

Beginnen wir mit dem Ansprechen der Tastatur. Normalerweise geschieht dies von BASIC-Programmen aus durch INPUT und GET. Ein Blick in das CBM-Handbuch verrät uns außerdem, daß sich die Tastatur mit der Gerätenummer 0 auch über OPEN erreichen läßt. Man kann dann durch die bekannten Peripheriebefehle wie auf eine Floppy oder eine Datasette zugreifen. Im Gegensatz zum normalen INPUT gibt dieser Befehl übrigens kein Fragezeichen aus. Dies gibt zur Vermutung Anlaß, daß auch die "Folterstrecke für Adler-Such-System-Tipper" (so ein Zeitgenosse) über eine Art Interface an den I/O-Bereich angeschlossen ist. Dieses Interface ist der CIA 1. An zwei parallelen Ports (die sehr nahe mit dem USER-PORT verwandt sind) wird die Abfrage vollzogen. Dazu sind die 64 Tasten elektrisch in 8 Zeilen und 8 Spalten aufgeteilt worden. Einer der beiden Ports ist auf Ausgabe programmiert. Hier wird die Spalte ausgegeben, die abgefragt werden soll. Ist eine Taste gedrückt worden, so wird dies in dem auf Eingabe geschalteten zweiten Port registriert. Die Interrupt-Routine, die für die Tastaturabfrage zuständig ist, hat also nichts weiter zu tun, als nacheinander alle 8 Spalten anzuwählen und die gedrückten Tasten festzustellen. Anhand einer Dekodiertabelle im ROM wird dann der ASCII-Code der Taste errechnet und dieser im Tastaturpuffer zwischengespeichert.

Läuft der Interpreter im Direktmodus, so holt er sich nach dem Interrupt den ASCII-Code aus dem Puffer ab und verarbeitet diesen dann (z.B. 13 = RETURN, bewirkt eine Befehlsausführung). Sollte gerade ein BASIC-Programm laufen, so wird der

Tastaturpuffer so lange unverändert bleiben, bis ein GET, INPUT oder Programmende auftritt. Bei GET holt der Interpreter einfach das erste Zeichen aus dem Puffer und speichert es in der im Befehl angegebenen Variablen ab. Ein INPUT funktioniert ähnlich, nur werden die Zeichen hier zusätzlich noch auf den Bildschirm geschrieben und die Abfrage so lange wiederholt, bis ein RETURN eingegeben wurde.

Die oben beschriebene Tastaturmatrix weist übrigens noch zwei Besonderheiten auf. So ist die RESTORE-Taste in dieser Matrix nicht enthalten. Sie wirkt direkt auf den Prozessor (ähnlich dem RESET-Taster aus Kap. 1.6.) und löst dort einen speziellen Interrupt aus. Diese Routine prüft, ob gleichzeitig die RUN/STOP-Taste gedrückt wurde. Ist dies der Fall, so wird eine Art MINI-RESET ausgeführt, sonst läuft alles normal weiter.

Die zweite Besonderheit stellen die SHIFT-Tasten dar. Der Rechner kann zwischen der linken und rechten Taste unterscheiden, da beide in verschiedenen Spalten liegen. Im Gegensatz dazu ist die SHIFT-LOCK-Taste aber nur eine besondere Form der linken SHIFT-Taste, d.h. zwischen beiden kann nicht unterschieden werden.

Das ganze Prinzip läßt sich übrigens mühelos auf den VC-20 übertragen, nur die elektrische Anordnung der Tasten ist anders.

9.2. Gleichzeitige Abfrage von zwei Tasten

Wir werden jetzt die Kenntnisse aus dem letzten Kapitel in praktische Anwendungen umsetzen. Für viele Programme ist es wünschenswert, mehrere Tasten gleichzeitig abfragen zu können, um z.B. zwei Raumschiffe unabhängig voneinander steuern zu können. Sehen wir uns dazu einmal die Tastaturmatrix (Abb. 9) an.

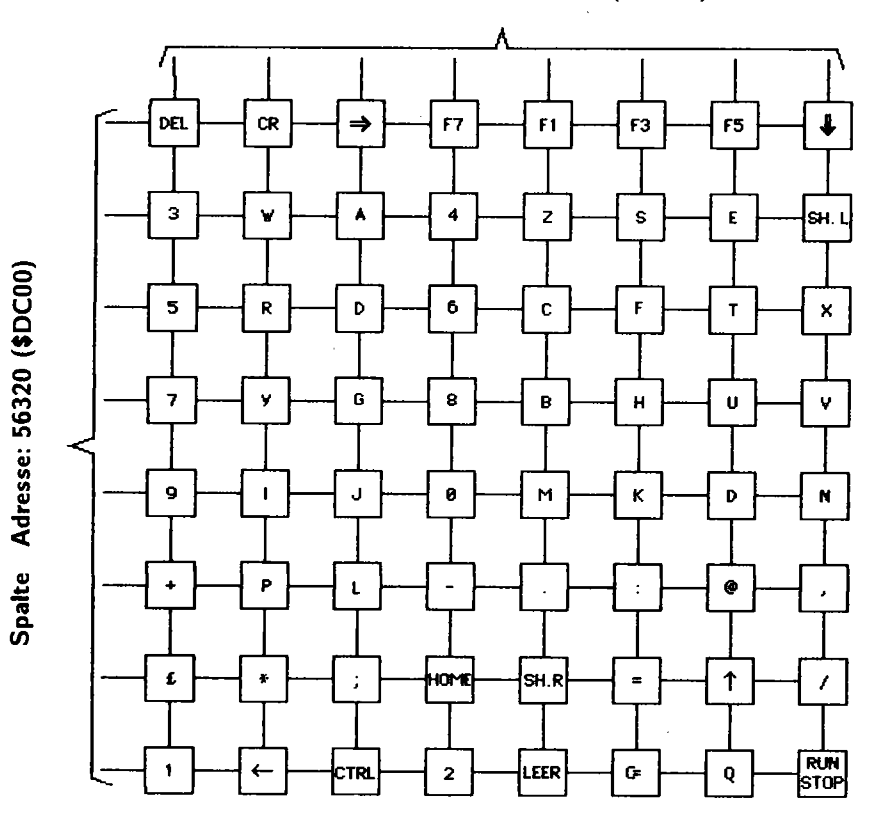


Abb. 9: Tastaturmatrix

Die beiden Speicherzellen, an denen die Tastatur angeschlossen ist, sind 56320 und 56321. Im Normalfall sind alle Bits dieser Register auf 1. Soll eine bestimmte Zeile angewählt werden, so muß das betreffende Bit in 56320 auf 0 gebracht werden. Ähnlich verhält es sich mit der Rückmeldung von der Tastatur. Ist eine Taste gedrückt, so wird das entsprechende Bit in 56321 auf 0 gesetzt.

Was die Interruptroutine kann, können wir schon lange. Durch einen POKE-Befehl können wir eine bestimmte Zeile auswählen und dann die Bits der abzufragenden Tasten testen. Sollten die beiden Tasten in verschiedenen Zeilen liegen, so fragen wir diese einfach nacheinander ab.

Da die Interruptroutine uns ins Handwerk pfuschen könnte (auch sie trifft ja eine Zeilenauswahl), schalten wir sie einfach ab. Außerdem muß die RUN-STOP-Taste mittels POKE 788, 52 gesperrt werden, da sonst jede Abfrage der untersten Tastaturzeile einen BREAK hervorrufen würde.

Alles zusammen ergibt dann diese Befehle:

```
POKE 56334, PEEK (56334) AND 254
POKE 788, 52
POKE 56320, Zeilencode
IF (PEEK (56321) AND (2^Bitnr.)=0)THEN PRINT "Taste
gedrückt"
POKE 56334, PEEK (56334) OR 1
POKE 788, 49
```

Damit läßt sich eine Taste abfragen. Sollen mehrere Tasten "beobachtet" werden, müssen noch weitere IF-THEN-Konstruktionen und ggf. auch weitere POKE-Befehle zur Zeilenauswahl eingefügt werden. Der Zeilencode berechnet sich nach dieser Formel:

```
CODE= 255-2^Zeilennr.
```

Die Zeilennummer stellt die Position des Bits innerhalb der Speicherzelle 56320 dar, das die gewünschte Zeile anwählt. Die

IF-THEN-Konstruktion in den oben genannten Befehlen hat die Aufgabe, zu testen, ob das gewünschte Spaltenbit auf 0 gesetzt wurde.

Eine andere Möglichkeit, zwei Tasten gleichzeitig abzufragen, bietet Speicherzelle 653. Hier wird das aktuelle SHIFT-Muster angezeigt, d.h., die Bits dieses Registers geben wieder, welche der drei Tasten SHIFT, COMMODORE und CONTROL gerade gedrückt werden. Für ein SHIFT wird Bit 0 auf 1 gesetzt, für C= Bit 2, für CONTROL Bit 3. Das Setzen der Bits geschieht unabhängig voneinander; sind alle drei Tasten gleichzeitig gedrückt, so werden auch alle drei Bits gleichzeitig gesetzt. Zuständig dafür ist wieder die Interruptroutine. Wollen wir die Speicherzelle 653 nutzen, darf der Interrupt nicht ausgeschaltet sein. Mit der Befehlsfolge

```
IF (PEEK (653) AND 2^Bitnr.) THEN PRINT "Taste gedrückt"
```

kann der Rechner vom BASIC aus feststellen, ob eine bestimmte Taste gedrückt wird.

Zusammenfassung: Gleichzeitige Tastaturabfrage

Es gibt zwei Möglichkeiten:

a. PEEK (653) gibt das SHIFT-Muster an. Damit können drei Tasten unabhängig voneinander getestet werden.
b. Nach Spaltenauswahl durch POKE 56320, X kann in Speicherzelle 56321 abgelesen werden, welche Taste in der Spalte gedrückt wurde. Die Anordnung ist in Abb. 4 aufgelistet.

9.3. Tasten sperren

Für viele Anwendungen ist es wünschenswert, einzelne Tasten (vor allem RUN/STOP) oder die ganze Tastatur zu sperren. Dafür gibt es beim 64er viele Möglichkeiten.

Um die gesamte Tastaturabfrage zu sperren, kann die Interruptroutine abgeschaltet werden. Es wird jetzt auch kein Cursor mehr erzeugt - der Rechner scheint sich aufzuhängen. Mit RUN/STOP-RESTORE kann dies jedoch aufgehoben werden.

Das gleiche gilt für POKE 649,0. Auch hier wird die Tastatur abgeschaltet, der Cursor kann jedoch weiterhin erzeugt werden (auch künstlich). Auch die RUN/STOP-Taste behält ihre Wirkung. Interessant ist die Entstehung dieser Sperre. Speicherzelle 649 gibt die Länge des Tastaturpuffers an. Wird jetzt die Länge auf 0 gesetzt (normal = 10), so meint das Betriebssystem, der Tastaturpuffer sei schon voll und vergißt deshalb die gedrückte Taste. Da das BASIC alle Tastendrücke (egal ob im Direktmodus oder per GET oder INPUT) über den Puffer holt, funktionieren keine Eingaben mehr.

Das gleiche Ergebnis liefert POKE 655,71. Damit wird der Zeiger auf die Tastaturdekodiertabelle geändert. Folge: Die Interruptroutine kann keine ASCII-Codes mehr bilden - der Tastaturpuffer bleibt leer. Wieder eingeschaltet wird mit POKE 655,72.

Will man nur die RUN/STOP-Taste abschalten, so kann der Befehl POKE 788, 52 benutzt werden. Danach kann ein BASIC-Programm nur noch per RUN/STOP-RESTORE gestoppt werden. Die BREAK-Funktion wird durch POKE 788, 49 wieder eingeschaltet.

Durch POKE 792, 193 wird der Mini-Reset (STOP & RESTORE) verhindert. Die STOP-Taste zeigt aber weiterhin Wirkung. Kombiniert man die letzten beiden POKEs, so läßt sich ein BASIC-Programm gar nicht mehr stoppen (außer mit der brutalen EIN-AUS-Methode). Soll das Programm zusätzlich noch vor LIST geschützt werden, so sollte POKE 808, 234 eingegeben werden. Danach sind alle BREAK-Möglichkeiten unwirksam und ein LIST liefert unsinnige Ergebnisse.

Zusammenfasssung: Tastatursperren

Ganze Tastatur abschalten:

1. Interrupt abschalten
2. POKE 649,0 (Tastaturpuffer auf Länge 0)
3. POKE 655,71 (Zeiger auf Dekodiertabelle ändern)

RUN/STOP aus: POKE 788, 52
RUN/STOP ein: POKE 788, 49
RESTORE aus: POKE 792, 193
RESTORE ein: POKE 792, 71
BREAK aus + Listschutz: POKE 808, 234

9.4. Die Repeatfunktion

Sie benutzen sie immer dann, wenn Sie mit dem Cursor an entfernte Stellen auf dem Bildschirm "fahren" wollen. Doch in den wenigsten Fällen haben Sie die Repeatfunktion bewußt wahrgenommen, und wenn, dann nur, wenn Sie sie mit anderen Tasten als der Cursorsteuerung benutzen wollten. Dieser Wunsch läßt sich erfüllen!

Speicherzelle 650 regelt den Umfang der Repeatfunktion. Im Normalfall steht in diesem Register eine 0. Das zeigt der Interruptroutine an, daß nur die Cursortasten und Space wiederholt werden sollen. Ist Bit 6 gesetzt (per POKE 650, 64), so wird die Repeatfunktion ganz abgeschaltet. POKE 650, 128 bewirkt genau das Gegenteil. Jetzt haben Sie auch auf den normalen Zeichentasten wie A, S, D etc. die Wiederholfunktion. Neben diesen primär nützlichen Details gibt es auch anderes Wissenswertes zum Repeat. Deshalb sei hier erklärt, wie die Interruptroutine überhaupt die Tastenwiederholung erzeugt. Bevor ein Tastendruck automatisch wiederholt wird, vergeht eine gewisse Vorlaufzeit (ca. 0,5 Sekunden). Dies soll verhindern, daß ein verfrühtes Repeat den Benutzer bei der Arbeit stört, wenn eine Taste zufällig etwas länger niedergedrückt wird. Diese Vorlaufzeit wird in Register 652 erzeugt. Die dort gerade stehende Zahl (meist 16) wird durch den Interrupt bis auf 0 heruntergezählt. Erst wenn die 0 erreicht ist, kann die Repeatfunktion starten. In

diesem Fall wird in ähnlicher Weise der Inhalt der Speicherzelle 651 heruntergezählt. Immer, wenn die 0 erreicht ist, wird ein neuer "Tastendruck" zusätzlich in den Puffer hereingeschoben und das Register mit einem neuen Startwert (4) geladen. Daher kann durch POKE 651, 255 die Repeatfunktion um ca. 4 Sekunden hinausgezögert werden.

Das ist schon das ganze Geheimnis um die Tastenwiederholung!

Zusammenfassung: Repeatfunktion

POKE 650, 128: Repeat auf alle Tasten
POKE 650, 64: Repeat abschalten
POKE 650, 0: Urzustand
POKE 651, 255: Repeat um 4 Sek. verzögern

9.5. Tastaturabfrage einmal anders

Wie Sie aus den letzten Abschnitten wissen, bringt die Interruptroutine die Tastendrücke als ASCII-Codes in den Tastaturpuffer. Auf dem Weg dahin gibt es aber eine Zwischenstation - die Speicherzelle 203. Hier wird der sogenannte Tastaturcode (siehe Tab. 7) zwischengespeichert, der als Zeiger innerhalb der Dekodiertabelle dient. Der Code erscheint so lange in diesem Register, wie die Taste gedrückt wird. Man kann daher mittels PEEK (203) z.B. zeitabhängige Eingaben programmieren, bei denen das Ergebnis von der Dauer des Tastendrucks abhängt. In Tabelle 7 finden Sie eine Übersicht über die Tastaturcodes, die leider nicht sehr viel mit den ASCII-Codes gemeinsam haben.

Ein Tastendruck, der durch PEEK (203) entdeckt wurde, ist dadurch noch nicht aus dem Tastaturpuffer gelöscht. Er kann durch GET oder INPUT in eine Variable gebracht werden. Damit kann man schon vor der eigentlichen Eingabe Daten überprüfen. Außerdem wird das Zwischenspeichern des Tastaturcodes nur durch das Abschalten des Interrupts verhindert. Eine Tastatursperre kann so eventuell umgangen werden.

Der Tastaturpuffer läßt sich übrigens auch löschen. Die Speicherzelle 198 gibt die Anzahl der schon gespeicherten ASCII-Codes an. Durch POKE 198,0 wird ein Löschen bewirkt, da jedes jetzt eintreffende Zeichen ein altes im Puffer überschreibt. Nach dem Löschen kann mit WAIT 198, 1 bis zum nächsten Tastendruck angehalten werden. Sobald ein neues Zeichen eintrifft, wird dies im Pufferzähler (also der Speicherzelle 198) registriert. Der WAIT-Befehl hat dabei nur die Aufgabe, die Ankunft des Zeichen zu erkennen und dann den Programmablauf wieder freizugeben. Danach kann das Zeichen per GET aus dem Puffer geholt werden. Man spart sich so umständliche IF-THEN-Konstruktionen.

Der Tastaturpuffer selbst liegt in den Speicherzellen 631 bis 640. Hier werden die Zeichen als ASCII-Codes abgelegt, die das BASIC sich dann abholen kann. Durch das EinPOKEn von Zeichen können Tastendrücke simuliert werden. Dabei muß allerdings auch der Zeiger in Speicherzelle 198 entsprechend erhöht werden, sonst "entdeckt" das BASIC diese Zeichen gar nicht.

Wie Sie jetzt sicher selbst feststellen, ist die Tastaturabfrage sehr vielseitig. Entwickeln Sie eigene Ideen, wie Sie sie nutzen können!

Zusammenfassung: Tastaturabfrage

PEEK (203) gibt den sog. Tastaturcode der gerade gedrückten Taste aus.
POKE 198, 0 löscht den Tastaturpuffer
POKE 198, 0: WAIT 198, 1 wartet auf einen Tastendruck
Die Speicherzellen 631 - 640 enthalten den Tastaturpuffer. Durch EinPOKEn von Codes können Tastendrücke simuliert werden.

Taste	Code	Taste	Code
A	10	3	8
B	28	4	11
C	20	5	16
D	18	7	24
E	14	8	27
F	21	9	32
G	26	CRSR_left	57
H	29	+	40
I	33	-	43
J	34	£	48
K	37	Home	51
L	42	Del	0
M	36	@	46
N	39	*	49
O	38	CRSR_up	54
P	41	:	45
Q	62	;	50
R	17	=	53
S	13	CR	1
T	22	,	47
U	30	.	44
V	31	CRSR_down	7
W	9	CRSR_right	2
X	23	F1	4
Y	25	F3	5
Z	12	F5	6
0	35	F7	3
1	58	Stop	63
2	59	Spc	60

Tab. 7: Tastaturcodes

10. Joystick, Paddles, Lightpen und anderes

Jeder kennt sie, aber kaum jemand weiß, wie sie funktionieren. Gemeint sind die Zusatzgeräte für Grafik- und Spielprogramme. Der Joystick ist am weitesten verbreitet, es gibt sogar Menschen, die sagen, ein C-64 ohne Joystick sei nicht vollständig. Nichts desto trotz folgen jetzt die Beschreibungen der einzelnen Zubehörteile, wobei sowohl Funktionsweise als auch Abfragetechniken aufgeführt sind.

10.1. Der Joystick

Viele wird es verblüffen, aber es stimmt. Für den 64er ist der Joystick eigentlich nur eine Art zweiter Tastensatz. Er wird nämlich über eine Tastaturspalte abgefragt. Die beiden Joystickports sind jeweils an der CIA 1 angeschlossen. Bei Port 1 wird das Register 56321 zur Rückmeldung benutzt. Die Joystickpositionen entsprechen den Tasten der Spalte 7. Soll also der Joyport 1 abgefragt werden, so muß Speicherzelle 56320 den Wert 127 enthalten. Dies ist immer dann der Fall, wenn die Interruptroutine die Tastaturabfrage beendet hat. Ist dagegen die Tastatur vorher über die Speicherzellen 56320/1 abgefragt worden, so sollte vor der Joystickbenutzung jeweils POKE 56320, 127 erfolgen.

Je nach Stellung des Steuerknüppels werden verschiedene Bits in 56321 gelöscht. Die Tabelle zeigt, welche Bits wofür zuständig sind:

Bit	4	3	2	1	0
Richt.	Feuer	rechts	links	unten	oben
Taste	Space	2	Ctrl	←	1

Unter den Belegungen sind die Tasten angegeben, mit denen der Joystick "simuliert" werden kann.

Ist die RUN/STOP-Taste nicht abgeschaltet, so kann man die Speicherzelle 145 zweckentfremden. Hier wird vom Betriebssystem eine Kopie der Speicherzelle 56321 erzeugt. Daher kann Joyport 1 auch per PEEK (145) abgefragt werden.

Etwas komplizierter verhält es sich mit Joyport 2. Er belegt die Speicherzelle 56320. Diese ist aber eigentlich für die Spaltenauswahl (also eine Ausgabe) vorgesehen. Die Abfrage des Joysticks verlangt aber eine Eingabe von außen. Also muß dieser Port des CIA umgeschaltet werden. Das kann durch POKE 56322, 224 erreicht werden. Dieser Befehl hat zwei Dinge zur Folge. Zum einen kann in Speicherzelle 56320 jetzt genau wie in Speicherzelle 56321 die Joystickbewegung abgelesen werden. Hinzu kommt aber auch eine Tastatursperre, die entweder durch RUN/STOP-RESTORE oder durch POKE 56322, 255 aufgehoben werden kann.

Die Funktionsweise eines Joysticks ist sehr einfach. Er besteht einfach aus 5 mehr oder weniger aufwendigen Tastern. Einer wird für den Feuerknopf benutzt, die anderen sind unter dem

Steuerknüppel in den vier verschiedenen Bewegungsrichtungen angebracht. Je nach Stellung des Knüppels wird dann der entsprechende Taster betätigt - der 64er registriert das dann in den genannten Speicherzellen.

Natürlich gibt es auch unter den verschiedenen Joysticks auf dem Markt Unterschiede. Einfache Exemplare (wie z.B. der Commodore-Joystick VC-1311) arbeiten mit einfachen Folienkontakten (ehemaligen ZX-81-Besitzern sicher noch in unguter Erinnerung), aufwendigere Verwandte dagegen besitzen Mikroschalter, die sich meist mit einem kleinen Klick bemerkbar machen.

Beim Kauf sollte darauf geachtet werden, daß der Joystick möglichst abgerundete Kanten besitzt. Andernfalls können beim Spiel sehr schnell Ermüdungserscheinungen auftreten. Im übrigen passen alle Atari-kompatiblen Joysticks auch für den Commodore.

Zusammenfassung: Joysticks

Joyport 1 abfragen: PEEK (56321)
Dabei muß Speicherzelle 56320 den Wert 127 enthalten
Joyport 2 abfragen: POKE 56322, 224: REM auf Port umschalten
PEEK (56320)
Port 1 kann auch hilfsweise über PEEK (145) abgefragt werden.

10.2. Paddles

Die Paddles sind allgemein auch als Drehregler bekannt. Ihre Aufgabe ist es im Gegensatz zum Joystick, der nur eine Richtung angibt, durch die Stellung des Reglers eine Position oder einen Wert an den Rechner zu übermitteln. Dazu ist ein Potentiometer eingebaut. Je weiter man es in die eine Richtung dreht, desto besser kann (vereinfacht gesagt) der Strom aus dem Rechner hindurchfließen, und umgekehrt. Der 64er kann über sogenannte AD-Wandler (Analog-Digital-Wandler) die ankommende

Spannung messen und das analoge Meßergebnis in eine digitale Zahl umwandeln. Diese Zahl kann dann in speziellen Registern abgelesen werden. Die AD-Wandler und diese Register sind Teil des SID. Da pro Joyport zwei Paddles (allerdings nur an einem Stecker) angeschlossen werden können, gibt es auch zwei Wandler und zwei Register. Deren Adressen sind 54297 und 54298. Beide Paddles haben auch je einen Feuerknopf. Diese können wie die Joystickrichtungen "Links" und "Rechts" in den Registern 56321 (für Port 1) und 56320 (für Port 2; hier bitte das Umschalten auf Eingabe nicht vergessen) abgefragt werden.

Der aufmerksame Leser wird es längst festgestellt haben - die Beschreibung ist noch nicht komplett. Wir können insgesamt 4 Paddles an unseren C-64 anschließen, doch es stehen nur zwei Wandler zur Verfügung. Daher muß es eine Möglichkeit geben, zwischen beiden Ports umzuschalten. Durch Setzen des Bits 7 in Speicherzelle 56320 wird die Übernahme der Messwerte auf Port 2 verlegt. Dies kann aber wieder nur geschehen, wenn der Interrupt uns dabei nicht stört. Also: Ausschalten.

Zusammenfassung: Paddles

Abfragen der Paddlewerte in Registern 54297 und 54298.

Knopfdrücke werden durch Joystickpositionen "Links" und "Rechts" repräsentiert.

Umschalten der AD-Wandler auf Port 2 durch Setzen des Bits 7 in Register 56320.

10.3. Der Lightpen

Wir kommen jetzt zu einem Wunderwerk der Technik - zumindest erscheint es Außenstehenden so. Wie schafft es ein so unscheinbares Gerät wie ein Lightpen (der deutsche Name Lichtgriffel klingt noch unscheinbarer), Punkte auf dem Bildschirm

zu setzen? Des Pudels bzw. Griffels Kern ist eigentlich gar nicht so kompliziert.

Das eigentliche Setzen der Punkte wird von einem Programm übernommen. Es funktioniert wie die Grafikroutine aus Kapitel 6. Für den Stift bleibt nur noch die Aufgabe, die Koordinaten für die Punkte zu liefern.

Diese werden in zwei Registern übergeben. Woher weiß der VIC aber, wo der Lightpen gerade auf den Schirm zeigt? Um diese Frage zu beantworten, braucht man Kenntnisse über den Aufbau eines Fernsehbildes. Wie Sie sicher schon bemerkt haben, besteht es aus einzelnen Zeilen. Ein Elektronenstrahl wird Zeile für Zeile über den Schirm geführt. Soll ein Punkt aufleuchten, so wird der Strahl eingeschaltet. Dies bringt eine spezielle Beschichtung auf dem Glas zum Leuchten. Soll der Punkt dunkel bleiben, so bleibt der Strahl aus. Das Abfahren des Schirms geschieht so schnell, daß unser Auge das als stehendes Bild wahrnimmt. Der Aufbau eines einzelnen Bildes dauert nur Bruchteile von Sekunden.

Wird jetzt der Lightpen auf den Bildschirm gehalten und wird er dabei vom Elektronenstrahl getroffen, so schickt er einen Stromimpuls zum VIC. Dieser sieht nach, welcher Punkt innerhalb des Schirmbildes gerade zum TV-Ausgang geschickt wird. Da der VIC das Videosignal erzeugt, kann er immer feststellen, welche Koordinaten gerade an der Reihe sind. Die X- und Y-Werte werden dann in den Registern 19 (53267) und 20 (53268) des VIC abgelegt, wo sie ein Programm abholen kann. Vom BASIC aus kann dies per PEEK geschehen.

Die dadurch erhaltenen Werte liegen in einem Bereich von 0 - 255. Durch einen einfachen Dreisatz kann man sie umrechnen und dann den entsprechenden Punkt setzen.

Zusammenfassung: Lightpen

Lightpen-Koordinaten werden in den VIC-Registern 19 (53267) und 20 (53268) übergeben. Damit kann eine entsprechende Grafikroutine veranlaßt werden, Punkte zu setzen.

10.4. Andere Zubehörteile

Sie werden in einschlägigen Zeitschriften sicher schon einmal ein sogenanntes Grafiktablett gesehen haben. Der Anwender kann darauf wie auf einem Blatt Papier zeichnen, das Bild erscheint dann auf dem Bildschirm in hochauflösender Grafik.

Es gibt verschiedene Funktionsprinzipien für solche Grafiktabletts. Gemeinsam ist jedoch allen, daß erkannt wird, an welcher Stelle sich der Finger, Stift o.ä. gerade befindet. Meist wird das dann als mehr oder minder starke Spannung an die Paddleeingänge geschickt. Dort kann der Rechner dann für alles weitere sorgen. Auch hier kommt man also nicht ohne die entsprechende Software aus.

Auch mit den Paddleeingängen zu tun hat eine spezielle Sorte von Joystick, die ich hier Proportionaljoystick nennen möchte. Sie liefern nicht nur die allgemeine Bewegungsrichtung, sondern eine genaue Positionsbestimmung, bestehend aus zwei Koordinaten. Dies wird durch ein X/Y-Potentiometer möglich gemacht. Eigentlich handelt es sich dabei um zwei Potentiometer in einem Gehäuse mit nur einem gemeinsamen Regler, eines für die X-, das andere für die Y-Richtung. Bewegt man den Steuerknüppel, so ändern sich die Werte, die die beiden Potentiometer liefern, entsprechend der Richtung. Auf diese Art und Weise kann man dann jeden Punkt des Bildschirms ansteuern.

Der Vorteil dieser beiden Geräte liegt darin, daß sie fertige Positionskoordinaten liefern. Damit kann man sich das langwierige Hin- und Herfahren mit den herkömmlichen Joysticks sparen.

11. Der User-Port

Der User-Port macht den C-64 zu einem sehr vielseitigen Instrument. Leider erwähnt das Handbuch die Handhabung und Programmierung nicht mit einer einzigen Silbe. Angesichts dieser sträflichen Mißachtung (auch seitens des BASICs) sollen hier aushilfsweise wenigstens die Grundtechniken der Programmierung besprochen werden.

11.1. Allgemeines über Schnittstellenbausteine

Wie auch die Tastatur und die Joysticks wird der User-Port über ein CIA betrieben, diesmal handelt es sich dabei um CIA 2. Die CIA sind sogenannte Schnittstellen- oder I/O-Bausteine. Das sind Chips, deren Aufgabe es ist, Daten von Peripheriegeräten zu empfangen, an diese zu senden und die Kommunikation mit dem Prozessor zu gewährleisten.

Im allgemeinen besteht ein solcher Baustein aus drei Elementen. Zum einen ist da eine Einheit für parallele Ports, die Sie von der Tastaturabfrage her bereits kennen. Dazu kommen noch eine Zeitgebereinheit (die Sie auch schon kräftig benutzt haben, allerdings ohne es zu wissen) und ein serieller Port.

Die folgenden drei Abschnitte sollen Ihnen die Funktionsweise dieser Elemente verdeutlichen.

11.1.1. Der serielle Port

Fangen wir beim einfachsten Teil an. Wie Sie wissen, verarbeitet der Computer alle Bytes parallel, d.h. die 8 Bits werden gleichzeitig bewegt, manipuliert etc. Eine serielle Schnittstelle bewegt die 8 Bits eines Bytes aber nacheinander über den Draht. Das geht zwangsläufig etwas langsamer als die parallele Übertragung, bietet aber den Vorteil, daß man keine 8 getrennten Datenlei-

tungen braucht, sondern nur eine. So können Daten z.B. über Telefon übertragen werden.

Die Arbeit des Schnittstellenbausteins besteht in der Umwandlung der verschiedenen Formate. Der Prozessor liefert die zu sendenden Bits parallel am Baustein ab, dieser schickt sie dann nacheinander und im richtigen Takt über den Draht.

Umgekehrt werden ankommende Bits wie Perlen auf die Schnur gereiht, bis ein Byte komplett ist. Dieses wird dann an den Prozessor übergeben.

Müßte der Prozessor diese Arbeiten selbst ausführen, so wäre der Datenaustausch über einen seriellen Bus sehr sehr langsam, da für jedes zu übertragende Bit mehrere Maschinenbefehle ausgeführt werden müßten.

Da ich der Meinung bin, daß sich eine serielle Schnittstelle nur in Maschinensprache wirklich effektiv programmieren läßt, werde ich die dazu nötigen Methoden nicht vorstellen. Überdies enthält das ROM des 64ers bereits die komplette Software, die zum Betrieb einer seriellen RS.232 (als Steckmodul für den User-Port erhältlich) nötig ist. Damit kann die Schnittstelle über OPEN 1,2 angesprochen werden.

11.1.2. Der Timer

Immer, wenn ein interner Zeitablauf zu regeln ist, tritt der Timer in Aktion. Man kann seine Register mit beliebigen Zeitwerten laden. Dieser Wert wird kontinuierlich heruntergezählt. Ist die 0 erreicht, schickt der Timer ein entsprechendes Signal an den Prozessor. Ein Beispiel für diese Art der internen Regelung stellt der Interrupt dar (Aha!). Der Timer wurde so programmiert, daß er in Abständen von 1/60 Sekunde Alarm schlägt und danach wieder von vorne anfängt. Der Prozessor reagiert auf einen solchen Alarm mit der Unterbrechung des Hauptprogramms und dem Anspringen der Interruptroutine - voila!

In diesem Zusammenhang sei noch erklärt, wie das Abschalten des Interrupts aus Kapitel 1 funktioniert. Das Bit 0 der Speicherzelle 56334 bestimmt, ob der für den Interrupt zuständige Timer gerade herunterzählt, oder ob er in seiner Arbeit innehält. Ist das Bit auf 0 (und genau das bewirkt der POKE-Befehl), so bleibt der Zeitgeber einfach stehen. Folge: Es werden keine Unterbrechungen mehr ausgelöst.

Außer diesem Trick sollten Sie sich nicht an die Timer im 64er heranwagen. In den meisten Fällen wird ein Experimentieren mit dem Aufhängen des Rechners enden.

11.1.3. Der parallele Port

Alle Schnittstellenbausteine für den 6502 bzw. 6510 haben eines gemeinsam: Die Art der Programmierung der parallelen Ports. Meistens besitzt ein Baustein gleich zwei solcher Ports, wie auch die CIAs.

Jeder dieser Ports verfügt über 8 Datenleitungen, die entweder auf Ausgabe oder Eingabe programmiert werden können. Dazu besitzt der Chip zwei spezielle Register. Das Datenrichtungsregister zeigt an, auf welchen Modus die einzelnen Leitungen geschaltet sind. Eine 1 bedeutet Ausgabe, eine 0 steht für Eingabe.

Diese Wahl hat übrigens einen besonderen Grund. Würde ein 0 für den Ausgabemodus stehen, so könnte es beim Einschalten des Computers dazu kommen, daß zufällig Impulse an Peripheriegeräte geschickt werden. Diese könnten dadurch unbeabsichtigt in Aktion treten und z.B. Daten auf einer Diskette zerstören.

Das zweite Register für den Port hat je nach Modus verschiedene Aufgaben. Für Eingabeleitungen fungiert es als Auffangbyte, d.h. hier kann sich der Prozessor die empfangenen Daten abholen.

Bei Ausgabeleitungen schreibt der Prozessor hierhin die Daten, die zum Peripheriegerät geschickt werden sollen. Um dem angesprochenen Gerät mitzuteilen, daß die Daten bereitliegen, gibt

es die sogenannten Handshakeverfahren. Hat der Prozessor sein Byte beim I/O-Baustein abgeliefert, so teilt dieser durch eine spezielle Handshakeleitung mit, daß der Ansprechpartner die Datenbits in sein Register übernehmen kann. Der Sender wartet mit dem nächsten Byte aber so lange, bis der Empfänger ebenfalls auf einer Handshakeleitung meldet, daß er mit der Datenübernahme fertig ist. Dabei kann das ganze Handshakeverfahren über nur eine, aber auch über 2 Leitungen ablaufen.

Neben diesen Funktionen bieten die Bausteine meist noch weitere Einrichtungen, z.B. zum Senden und Empfangen von Impulsen. Auch muß die Datenübertragung nicht unbedingt nach dem Handshakesystem ablaufen.

11.2. Wie benutze ich den User-Port?

Der User-Port stellt uns einen parallelen Port und verschiedene "Zubehörleitungen" zur Verfügung. Die meisten dieser Leitungen liefern jedoch intern bereits genutzte Signale. So werden wir uns auf einen 8 Bit breiten Port und eine "ausgeliehene" Steuerleitung beschränken. Ausgeliehen deshalb, weil sie eigentlich vom Port A des CIA 2 stammt, also eine Datenleitung darstellt.

Der CIA 2 hat die Basisadresse 56576. Das ist auch die Adresse des Datenregisters für Port A (Reg. 0), wo Bit 2 den Zustand der Steuerleitung wiedergibt. Alle anderen Leitungen dieses Ports werden intern genutzt, deshalb darf nur Bit 2 manipuliert werden!

Anders verhält es sich mit Register 1 (56577). Das ist das Datenregister für Port B, der den eigentlichen User-Port darstellt. Hier sind alle acht Leitungen frei verfügbar.
Die Datenrichtungsregister folgen dann unter den Nummern 2 (56578 für Port A; Achtung, nur Bit 2 verändern!) und 3 (56579 für Port B). Diese werden in der bereits beschriebenen Weise benutzt.

Mit POKE 56579, 255 werden also alle 8 Datenleitungen auf Ausgabe programmiert, POKE 56579, 0 setzt sie wieder auf Eingabe.

Für die Steuerleitung muß diese Programmierung etwas vorsichtiger erfolgen. POKE 56578, PEEK (56578) AND 251 schaltet auf Eingabe, POKE 56578, PEEK (56578) OR 4 bewirkt das Gegenteil.

Um Daten auf dem Port auszugeben, schreiben wir diese einfach in Speicherzelle 56577. Umgekehrt können wir die ankommenden Daten direkt auslesen.

Den Strom auf der Steuerleitung können wir mit POKE 56576, PEEK (56576) OR 4 einschalten, ausgeschaltet wird mit POKE 56576, PEEK(56576) AND 251. Setzen wir beide Befehle direkt nacheinander ins Programm, so kann damit ein kurzer Impuls erzeugt werden.

Die Steuerleitung belegt den Pin M des User-Ports (siehe Abb. 2 oder CBM-Handbuch), die 8 Datenleitungen befinden sich an den Pins C bis L.

Zusammenfassung: Programmierung des User-Ports

Datenrichtungsregister für 8 Datenleitungen: 56579
Datenrichtungsregister für Steuerleitung: 56578 (nur Bit 2)
Datenregister für Port: 56577
Datenregister für Steuerleitung: 56576 (nur Bit 2)

11.3. Anwendungsbeispiele

Der User-Port läßt sich sehr vielseitig einsetzen. Daher sollen hier auch keine Beispielprogramme vorgestellt werden, sondern nur ein paar Anregungen für eigene Entwicklungen. Das einfachste Beispiel stellen Lampen oder LEDs dar, die evtl. über Treibertransistoren oder Relais an den User-Port angeschlossen und geschaltet werden. Damit läßt sich z.B. eine Lichtorgel re-

alisieren, die die Lautstärke der Musik über ein am AD-Wandler angeschlossenes Mikrofon mißt und dementsprechend die Lampen ein- und ausschaltet. Mit einem anderen Programm könnten ein Lauflicht oder weitere Effekte erzeugt werden.

Denkbar ist auch die Koppelung von zwei Commodore-Computern (egal welchen Typs, da sie alle einen User-Port besitzen), um Daten auszutauschen. Auf diese Weise könnte z.B. ein VC-20 Messungen vornehmen, die der 64er gleichzeitig auf seinem größeren Bildschirm in hochauflösender Grafik darstellt.

Elektronisch begabte Leser könnten sich auch an den Bau einer eigenen seriellen Schnittstelle machen, um damit z.B. Daten über Telefon fernzuübertragen. Ebenfalls denkbar ist auch der Anschluß eines Nicht-Commodoredruckers an den C-64. Auch dafür geeignet sind Fernschreiber, Lochstreifenstanzer und -leser, Home-Roboter oder Taschenrechner. Dem Erfindungsreichtum des Bastlers sind keine Grenzen gesetzt.

12. BASIC & Betriebssystem

Das Betriebssystem und auch das BASIC stellen uns viele Funktionen zur Verfügung, die nicht mit einem der in den vorherigen Kapiteln beschriebenen Details zusammenhängen. Oft ist es jedoch wünschenswert, diese Funktionen (z.B. List) zu beeinflussen, um bestimmte Zwecke zu verfolgen. Von diesen Manipulationsmöglichkeiten soll hier die Rede sein.

12.1. Erzeugen von BASIC-Zeilen per Programm

Stellen Sie sich vor, Sie wollten ein Programm schreiben, das den Graphen einer beliebigen Funktion in Hochauflösung auf den Bildschirm zeichnet. Wenn das Programm die Funktion nicht fest vorgeben soll, muß es eine Möglichkeit geben, den Term einzutippen. Für einfachere Versionen reicht es, wenn der Benutzer vorher in einer speziellen Programmzeile die Funktion per DEFFN selbst ins Programm einbaut. Doch dazu braucht der Benutzer Programmierkenntnisse. Bequemer wäre es, die Rechenvorschrift über INPUT einzugeben. Doch was nützt uns ein String, in dem ein Term gespeichert ist - ausgeführt werden kann er nicht. Die letzte Möglichkeit wäre, den Rechner sich selbst programmieren zu lassen. Das geht sogar sehr einfach.

Um die Methode zu verstehen, sollten wir zunächst einen Blick auf die normale Entstehung einer Programmzeile werfen. Alles beginnt damit, daß ein Anwender eine (hoffentlich) durchdachte Folge von Buchstaben und Zeichen eintippt. Diese Zeichen erscheinen gleichzeitig auf dem Bildschirm. War eines dieser Zeichen ein RETURN, so übernimmt der BASIC-Interpreter die gesamte Bildschirmzeile (nicht nur die eingetippten Zeichen) in den BASIC-Eingabepuffer und wandelt die Zeichenfolge in eine Programmzeile oder (wenn keine Zeilennummer am Anfang stand) in direkt ausführbare Befehle um. Dem Interpreter ist es also egal, ob die Zeichen eingetipppt oder etwa gePRINTet wurden. Darauf baut unsere Methode auf. Zunächst wird der beabsichtigte Text der Programmzeile auf dem Bildschirm ausgegeben. Dann müssen wir nur noch die Umwandlung in eine Pro-

grammzeile veranlassen. Dazu wird ein künstlicher Tastendruck erzeugt, indem der ASCII-Code in den Tastaturpuffer gePOKEd wird. Folgt jetzt im Programm ein END, so werden diese Tastendrücke nach dem Programmabbruch ausgeführt. Dabei ergeben sich zwei Probleme. Durch die Erzeugung einer neuen Zeile werden die Variablen gelöscht (wie auch bei der normalen Programmeingabe). Daraus folgt, daß die Erzeugung künstlicher Zeilen erfolgen sollte, wenn keine wichtigen Daten angefallen sind, also am Programmanfang. Müssen einige Variablen erhalten werden, so empfiehlt es sich, diese in freie RAM-Bereiche einzuPOKEn, die vom Betriebssystem nicht benutzt werden.

Zusätzlich soll das Programm nach der Zeilenerzeugung weiterlaufen. Daher muß nach der Zeile ein künstliches GOTO xxx stehen, das nach dem gleichen Muster wie die Zeile erzeugt wird. Hier ein Beispiellisting:

```
10 INPUT "Term: Y="; A$: REM Eingabe Funktionsterm
20 PRINT "(CLS)(3xCRSR DOWN)100 DEFFNF(X)="; A$: REM Zeile
ausgeben
30 PRINT "GOTO 70(HOME)";: REM Befehl zur Programmfortsetzung
40 POKE 631, 13: POKE 632, 13: REM 2 x RETURN
50 POKE 198, 2: REM Tastaturpuffer initialisieren
60 END
70 ...
```

Wenn Sie dieses Programm eingetippt und gestartet haben, werden Sie sehr schnell den Sinn der einzelnen Anweisungen verstehen, vor allem was die Bildschirmausgabe betrifft. Die erzeugte Zeile unterscheidet sich nicht von einer normal eingegebenen Zeile. Das Programm kann beliebig oft durchlaufen werden. Sollte eine fehlerhafte Eingabe gemacht worden sein, so quittiert der Interpreter dies mit einem SYNTAX-ERROR nach dem Durchlauf der neuen Zeile.

Diese Anwendung läßt sich übrigens noch stark erweitern. So können auf diese Weise auch Programmzeilen gelöscht werden, die man nicht mehr benötigt. Auch können mehrere Zeilen

gleichzeitig erzeugt werden. So ist die Eingabe ganzer Unterprogramme per INPUT möglich.

12.2. Listschutz

Bei Programmen, die auf persönliche Daten zugreifen, empfiehlt es sich, eine Codewortabfrage einzubauen. Damit das Codewort nicht durch LIST aufgedeckt werden kann, sollte die betreffende Programmzeile geschützt werden. Dies kann durch einen POKE-Befehl erreicht werden.

Zum Verständnis ist es nötig, das Format einer Programmzeile im Speicher zu kennen. Die ersten beiden Bytes einer Zeile bilden den Zeiger auf die nächste Zeile. Damit kann sich der Interpreter von Zeile zu Zeile "hangeln". Sind diese beiden Bytes 0, so ist danach keine Programmzeile mehr gespeichert; hier befindet sich also das Programmende.

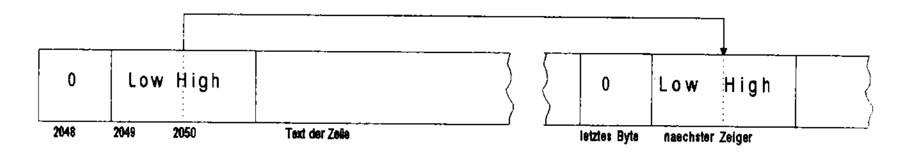


Abb. 10: Format von BASIC-Zeilen

Nach dem Zeiger folgen zwei Bytes mit der Zeilennummer. Auch diese ist wie ein Pointer aufgebaut. Dann folgen die Befehle im Interpretercode. Das Zeilenende wird durch eine Null repräsentiert. Mit dieser 0 können wir den Interpreter ein wenig hereinlegen. POKEn wir nämlich direkt nach der Zeilennummer eine 0 ein, so meint die LIST-Routine, die Zeile wäre bereits abgeschlossen und holt sich die nächste Programmzeile (der Pointer am Zeilenanfang blieb ja unverändert). Auch ein GOTO wird dadurch nicht beeinflußt, da die Routine, die eine bestimmte Zeile im Text sucht, sich ebenfalls an diesen Pointern orientiert. Die Routine, die den nächsten Befehl im Programm

sucht, tut dies aber nicht, sondern überspringt nach einer 0 einfach 4 Bytes. Deshalb "fehlen" die ersten vier Bytes der Zeile beim Programmablauf. Um die Ausführung der Befehle nicht zu behindern, müssen beim Schreiben der Programmzeile 5 beliebige Zeichen (aber kein Befehlswort!) eingefügt werden. Das erste dieser Zeichen wird durch die 0 überschrieben, die restlichen vier dienen als Platzhalter.

Woher wissen wir aber, welches Byte wir überschreiben müssen? Nun, auch dafür gibt es einen Trick. Wir bauen vor der zu schützenden Zeile einen STOP-Befehl ein und lassen das Programm bis hierhin ablaufen. Nach dem BREAK steht in den Speicherzellen 61 und 62 der Pointer auf dem nächsten BASIC-Befehl. Wenn der Stop-Befehl am Ende der Zeile steht, zeigt der Pointer auf das Zeilenende, also auf eine Null. Addiert man zu dieser Adresse noch 5 dazu, so erhält man das gewünschte Byte. Also frisch ans Werk mit POKE AD, 0. Nach diesem Befehl erscheint beim LISTen nur noch die Zeilennummer, der Text wird nicht mehr gezeigt. Es bleibt nur noch, den STOP-Befehl (der jetzt überflüssig ist) zu löschen. Hier noch einmal die Zusammenfassung der einzelnen Schritte:

1. Vor Zeile STOP einfügen.
2. In Zeile 5 Platzhalter (beliebige Zeichen) einfügen.
3. AD= PEEK (61) + 256*PEEK (62) + 5
4. POKE AD, 0
5. STOP-Befehl löschen.

Wollen Sie das ganze Programm vor LIST schützen, so bietet es sich an, den Vektor auf die LIST-Routine in der Zeropage zu verändern. Dadurch findet der Rechner sein Unterprogramm nicht wieder. Der Vektor steht in den Speicherzellen 774/775. Durch POKE 775, 1 wird dieser Zeiger derart "umgebogen", daß jeder LIST-Befehl wie RUN/STOP-Restore wirkt. Dieser Listschutz kann durch POKE 775, 167 aufgehoben werden.

12.3. Renumber

Besitzern einer BASIC-Erweiterung wie EXBASIC oder SIMONS BASIC ist er in guter Erinnerung: Der Renumberbefehl. Damit kann das im Speicher stehende Programm umnumeriert werden, was z.B. bei MERGE sehr vorteilhaft sein kann. Dieser Befehl kann aber auch ohne Basic-Erweiterung simuliert werden.

Wie Sie aus dem letzen Abschnitt wissen, beginnt jede Programmzeile im Speicher mit 2 Pointern. Der erste zeigt auf den Beginn der nächsten Zeile, der zweite verdient den Namen Zeiger eigentlich nicht, da er nur die Zeilennummer im Pointerformat angibt. Addieren wir zum Zeiger auf die nächste Zeile noch 2 hinzu, so erhalten wir die Adresse der nächsten Zeilennummer. Auf diese Weise können wir alle Programmzeilen abklappern und durch POKE die Zeilennummer ändern. Hier das Programm dazu:

```
63900 BA= PEEK (43) + 256 * PEEK (44)
63910 INPUT "Startnummer"; SA: INPUT "Schrittweite"; SW
63920 HI= SA/256: LO= SA - HI * 256
63930 A= PEEK (BA+2) + 256 * PEEK (BA+3)
63940 IF A größer-gleich 63900 THEN PRINT "OK!": END
63950 POKE BA+2, LO: POKE BA+3, HI
63960 BA= PEEK (BA) + 256 * PEEK (BA+1): SA= SA+SW
63970 PRINT SA "=" A: GOTO 63920
```

Diese Routine wird an das umzunumerierende Programm angehängt (entweder eintippen oder MERGEn) und dann mit RUN 63900 gestartet. Die hohen Zeilennummern wurden gewählt, damit es immer am Ende des Programms steht.

Zeile 63900 berechnet die Basisadresse der ersten Zeile aus dem Pointer auf den BASIC-Start. Bei Durchlauf der Zeile 63910 gibt der Benutzer ein, mit welcher Startnummer begonnen und mit welchem Zeilenabstand das Programm umnumeriert werden soll. Wenn Sie möchten, daß das Programm mit Zeile 10 beginnt und

der übliche Zehnerabstand eingehalten werden soll, geben Sie hier zweimal 10 ein.

Zeile 63920 berechnet High- und Lowbyte der neuen Zeilennummer, Zeile 63930 holt die alte Zeilennummer aus dem Speicher. Ist diese größer-gleich 63900, so wird der Renumbervorgang abgebrochen, da die Routine sich nicht selbst umnumerieren darf.

Die nächste Zeile POKEd High- und Lowbyte der neuen Nummer ein.

Schließlich wird noch die Basisadresse der nächsten Zeile berechnet, die Zeilennummer um die Schrittweite erhöht und ein Umnumerierungsprotokoll ausgegeben. Dieses Protokoll zeigt für jede neue Zeilennummer das alte Äquivalent an. Damit wird das Anpassen der GOTO, GOSUB und sonstiger Sprungbefehle erleichtert. Unsere Routine kann nämlich die Sprungadressen innerhalb des Programmtextes nicht ändern. Besitzern eines Druckers empfehle ich, die Protokollausgabe umzuleiten, damit man die Vergleichstabelle hinterher schwarz auf weiß vor sich hat. Zu diesem Zweck sollte am Anfang der Routine OPEN 1,4: CMD 1 eingefügt werden.

12.4. RENEW

Der NEW-Befehl führt von allen BASIC-Befehlen am häufigsten zu Tobsuchtsanfällen bei Computerbesitzern. Denn eines haben alle Computer gemeinsam. Durch das unbedachte Eintippen dieser drei Buchstaben (+ RETURN) hat sich schon mancher Programmierer ungewollt um die Früchte harter Arbeit gebracht, weil er vergessen hatte, das Programm vorher zu speichern. Um gegen solche Schicksalsschläge gewappnet zu sein, habe ich einen Trick ausgegraben, der die NEW-Katastrophe wieder rückgängig machen kann.

Unter Commodore-Programmierern ist es längst kein Geheimnis mehr, daß durch NEW der Speicher nicht etwa vollständig gelöscht wird, sondern nur die beiden zentralen Stützzeiger des

Programms zurückgesetzt werden. Der erste dieser beiden ist der Pointer auf den Beginn der Variablen bzw. das Programmende. Nach dem NEW zeigt er auf den Programmanfang, was dazu führt, daß alle Variablen oder neuen Programmzeilen, die jetzt benutzt werden, das alte Programm (das immer noch im Speicher stand) überschreiben. Deshalb oberstes Gebot nach einem versehentlichen NEW: Keine Befehle oder Zeichen eingeben, die nichts mit RENEW zu tun haben! Auch ein einfacher Buchstabe + RETURN erzeugt schon eine Variable, obwohl der Rechner einen Syntax Error ausgibt!

Der zweite Zeiger befindet sich in der ersten Programmzeile, genauer gesagt an den Adressen 2049 und 2050. Normalerweise zeigt er auf die nächste Zeile, jetzt aber enthält er zwei Nullen, um das Programmende zu markieren. Für RENEW bleiben also zwei Dinge zu tun:

1. Ende der ersten Zeile suchen. Dieses wird durch eine 0 markiert. Ist diese Null gefunden, so muß deren Adresse + 1 als Zeiger in die Bytes 2049 und 2050 gePOKEd werden.

2. Programmende suchen. Das Programmende läßt sich durch eine Null im Highbyte des Zeigers auf die nächste Zeile erkennen. Ist das Ende gefunden, so wird die Adresse um 2 erhöht, was den Beginn des Variablenbereiches angibt. Damit kann der Zeiger entsprechend geladen werden.

Beide Aufgaben können weitgehend von ROM-Routinen übernommen werden. Folgende Anweisungen sind dazu einzutippen:

```
POKE (PEEK(43)+256*PEEK(44)),1
SYS 42291: POKE 252, PEEK(35): POKE 251, PEEK(781)
POKE 46, (PEEK(251)+256*PEEK(252)+2)/256
POKE 45, (PEEK(251)+256*PEEK(252)+2)-PEEK(46)*256:
CLR
```

Tippen Sie alles bitte genau wie abgedruckt ein! Wichtig ist vor allem, daß Sie die vorgegebene Zeilenstruktur beibehalten.

Mit dem POKE-Befehl wird dem BASIC vorgegaukelt, der Pointer in der ersten Zeile wäre noch nicht gelöscht. Der darauffolgende SYS-Befehl ruft die ROM-Routine zum Neuberechnen der Zeilenpointer auf. Dieses Unterprogramm klappert nach und nach alle BASIC-Zeilen ab und berechnet ihre Zeiger neu. Ist die letzte Zeile erreicht, so bleibt in internen Registern der Zeiger auf das letzte Programmbyte stehen. Daraus berechnet die nächste ROM-Routine den Zeiger auf den Beginn der Variablen (45/46). Fertig!

Bevor Sie diesen Trick anwenden, sollten Sie allerdings eines beachten. Der POKE-Befehl muß immer in das Byte direkt nach dem Programmstart gehen. Ist der Zeiger dafür (Bytes 43/44) z.B. durch einen RESET verändert worden, so muß er vor dem RENEWen wieder auf die alten Werte gesetzt werden.

12.5. RESTORE

Der RESTORE-Befehl wird nicht sehr oft benutzt. Geschieht es doch einmal, daß man den DATA-Zeiger zurücksetzen muß, so ist es meist sinnvoller, eine Zeilennummer oder gar das Data-Element selbst angeben zu können, um den Zeiger nicht auf weiter vorn stehende Daten, die nicht mehr gebraucht werden, zurücksetzen zu müssen. So ließe es sich vermeiden, daß die nicht benötigten Daten jedesmal überlesen werden müssen, bevor man auf das eigentliche Datum zugreifen kann.

Mit ein paar POKE-Befehlen kann jedoch auch hier Abhilfe geschaffen werden. Dazu muß man wissen, daß der Interpreter sowohl die Zeilennummer als auch die Adresse des letzen DATA-Elements in der Zeropage speichert. Die Zeilennummer ist als Bytepaar (pointerähnlich) in den Speicherzellen 63/64 abgelegt, die Adresse des Bytes nach dem letzten DATA-Element, das gelesen wurde, finden wir in 65/66.

Wollen wir jetzt ein RESTORE simulieren, so können wir folgendermaßen vorgehen:

1. DATAs bis zum Element vor dem gewünschten Ziel lesen lassen (z.B. im Direktmodus). Soll auf das 5. Element zurückgesetzt werden, so müssen also die ersten 4 DATAs gelesen werden.

2. PRINT PEEK (63), PEEK (64)
   Die erscheinenden Zahlen repräsentieren die Zeilennummer. Zahlen bitte merken!

3. PRINT PEEK (65), PEEK (66)
   Auch diese Zahlen müssen wir uns merken! Sie bilden den Zeiger auf das Byte nach dem letzten DATA-Element. Bis hierhin müssen alle Befehle vor dem eigentlichen Programmablauf gegeben werden.

4. POKE 63, 1. Zahl: POKE 64, 2. Zahl
   POKE 65, 3. Zahl: POKE 66, 4. Zahl

Diese Befehle werden statt RESTORE ins Programm an die Stelle eingebaut, an der der Datazeiger zurückgesetzt werden soll. Dadurch werden die Pointer auf den Stand gebracht, den Sie vor dem Lesen des gewünschten Elements hatten. Für das BASIC entsteht der Eindruck, als hätte es die nachfolgenden DATA-Zeilen noch nicht gelesen.

Allerdings hat diese Methode einen Nachteil. Nach jeder Änderung in Programmzeilen, die vor der gewünschten Position des Zeigers liegen, ändert sich die Adresse, die im Data-Pointer stehen sollte, da das BASIC den gesamten Programmtext im Speicher verschiebt. Deshalb sollten solche DATA-Blöcke ganz am Anfang des Programms vor den eigentlichen Befehlen stehen.

Zusammenfassung: RESTORE

Zeilennummer des letzten DATA-Elements ist in den Speicherzellen 63 und 64 gespeichert. Die Adresse des Bytes nach dem

letzten Element befindet sich als Zeiger in den Bytes 65 und 66. Beide Pointer können durch POKE verändert werden (vorher gewünschte Pointerwerte feststellen).

12.6. Verschiedene Tricks

Nach einer Programmunterbrechung oder einem Error zeigt der Rechner an, in welcher Zeile das Programm verlassen wurde. Hat man etwas voreilig den Bildschirm gelöscht, so erfährt man diese Zeilennummer meist nicht mehr. Hier schaffen die Speicherzellen 59 und 60 Abhilfe. Hier wird (im Zeigerformat) die letzte Zeilennummer abgelegt, die man sich durch

PRINT PEEK (59) + 256 * PEEK (60) ausgeben lassen kann.

Ein Programm vor SAVE schützen kann man mit dieser Sequenz:

POKE 801, 0: POKE 802, 0: POKE 818, 165

Dadurch werden die Vektoren, die der SAVE-Befehl benötigt, so umgebogen, daß kein Abspeichern mehr möglich ist. Nachteil: Schon durch einfaches Drücken von RUN/STOP-RESTORE hängt sich der Rechner auf.

Schließlich noch einige SYS-Befehle, die sich gut in eigenen Programmen verwenden lassen:

SYS 65499 setzt den TI$ auf 000000. Das geht schneller als die Zuweisung eines neuen Strings.

Ästheten unter den Commodore-Besitzern können ein Programm durch SYS 42115 (statt END) beenden. Damit wird ein Warmstart des BASICs bewirkt, was nichts anderes heißt, als daß das BASIC in den Direktmodus umschaltet. Dabei wird aber kein Ready ausgegeben; der Cursor steht sofort in der nächsten Zeile. Auch ein CONT bleibt nach SYS erfolglos.

Soll das Programm mit dem Einschaltbild beendet und gleichzeitig gelöscht werden, so bietet sich SYS 58253 an. Und einen künstlichen SYNTAX ERROR erzielt man durch SYS 44808.

Zusammenfassung: Tricks zum Betriebssystem

Letzte Zeilennummer ist in Speicherzellen 59 und 60 gespeichert.

SAVE-Schutz: POKE 801, 0: POKE 802, 0: POKE 818, 165

TI$ auf 0 setzen: SYS 65499
End ohne Ready: SYS 42115
Einschaltbild: SYS 58253
Syntax Error: SYS 44808

12.7. BASIC-Erweiterungen

Fast jeder Commodore-Besitzer kennt BASIC-Erweiterungen zumindest aus Anzeigen, wenn er nicht sogar selbst ein solches Programm besitzt. Die ersten Exemplare dieser nützlichen Helfer gab es schon zu den Zeiten des seligen PET (oder CBM 2000). Zunächst enthielten sie nur sogenannte Toolkit-Befehle, die das Editieren von Programmen erleichtern. Darunter fällt zum Beispiel AUTO. Dieser Befehl gibt automatisch die Zeilennummern im gewählten Abstand (z.B. 10) für die einzugebenden Programmzeilen vor, so daß man sich diese Tipparbeit sparen kann. FIND findet bestimmte Ausdrücke im Programmtext, RENUMBER, MERGE und RENEW kennen Sie bereits aus den vorhergehenden Kapiteln. Mit DEL können Sie ganze Programmteile löschen. TRACE ermöglicht durch Ausgabe der durchlaufenen Zeilen eine einfache Überwachung des Programmablaufes beim Testen. DUMP gibt alle benutzten Variablen samt Inhalt aus.

Komfortablere Versionen ermöglichen das Auffangen von Errors. Damit wird zum Beispiel die Korrektur von fehlerhaften Eingaben ermöglicht, ohne daß ein TYPE-MISMATCH-ERROR erscheint.

Seltener findet man die Möglichkeit, die Funktionstasten mit Zeichenfolgen zu belegen.

Da das BASIC des 64ers die phantastischen Sound- und Grafik-Möglichkeiten nicht unterstützt, bieten viele BASIC-Erweiterungen auch hier Befehle zum Zeichnen und zum Programmieren von Tonfolgen.

Einige Programme stellen auch Strukturierungsbefehle zur Verfügung, mit denen man Programme ohne GOTO-Befehle schreiben kann. In diesem Fall werden die einzelnen Programmteile in sogenannten Moduln (ähnlich Unterprogrammen) programmiert. Statt GOSUB wird jetzt z.B. mit CALL PLOT X,Y aufgerufen, um eine Punktsetzroutine zu erreichen. Diese Technik fördert die Übersichtlichkeit eines Programmes sehr.

Verbreitet ist auch der Einbau von speziellen DOS-Befehlen, die es z.B. ermöglichen, eine Directory direkt auf den Bildschirm zu holen, ohne ein Programm im Speicher zu löschen.

12.8. Andere Programmiersprachen

Der Commodore 64 zeichnet sich auch dadurch aus, daß es möglich ist, andere Programmiersprachen als das BASIC zu laden und damit zu arbeiten. Am bekanntesten ist hier wohl PASCAL. Sein Grundkonzept ist die strukturierte Programmierung, d.h. GOTO ist verpönt (einige PASCAL-Versionen beinhalten diesen Befehl gar nicht erst), Modularität ist Trumpf. Das soll verhindern, daß der Programmierer einfach drauflostippt, statt sich vorher ein Konzept auszuarbeiten. PASCAL wird immer als Compiler geliefert, d.h. vor dem Programmablauf wird der Programmtext zunächst in eine computer-freundliche Version (meistens Maschinensprache oder eine schnelle Zwischensprache) übersetzt.

Im Gegensatz dazu stellt FORTH eine Interpretersprache dar. Auch hier wird auf Strukturierung Wert gelegt, ja man kommt gar nicht drumherum, weil man sich eigene Befehle (allerdings nicht in Maschinensprache) definieren muß. FORTH stellt nur

wenige Grund-Befehle zur Verfügung und steht damit der Maschinensprache sehr nahe. Daraus resultiert auch eine sehr hohe Geschwindigkeit.

Ebenfalls sehr weit verbreitet ist LOGO. Diese Sprache ist so einfach zu erlernen, daß sogar Erstkläßler damit umgehen können. Hauptmerkmale: Turtle-Grafik (man bewegt eine gedachte Schildkröte wie einen Zeichenstift über den Bildschirm und kann damit sehr einfache Grafiken programmieren) und Modularität. LOGO eignet sich besonders für mathematische und geometrische Probleme.

12.9. Ein kleines Bonbon

Kennen Sie das? Sie haben gerade ein Listing ausgedruckt; eigentlich fehlt jetzt nur noch eine kleine Bedienungsanleitung, um die Programmdokumentation vollständig zu machen. Ärgerlich nur, daß man für die 3 Zeilen extra ein Textverarbeitungsprogramm laden muß.

Hier kann Ihnen ein kleiner Trick Linderung verschaffen. Durch POKE 22,35 wird das Betriebssystem derart verbogen, daß beim Listen eines Programms die Zeilennummern weggelassen werden. Schiebt man ein OPEN 1,4: CMD1 vor, so funktioniert das ganze auch auf dem Drucker. Auf diese Art und Weise kann man auf die schnelle ein paar Zeilen in Textform drucken lassen, indem man den Text einfach als Programmzeile eingibt.

Abgeschaltet wird dieses Phänomen durch POKE 22,25

13. Einführung in die Maschinensprache

In vielen Publikationen finden Sie immer wieder Progamme zum Abtippen. Sehr oft sind diese Programme in Maschinensprache geschrieben, einer Sprache, die dem Neuling wie ein Buch mit 7 Siegeln erscheint. Zugegeben, die Maschinensprache ist nicht so leicht zu erlernen wie BASIC, doch dafür ist sie sehr viel schneller und bietet dem erfahrenen Programmierer ungleich mehr Möglichkeiten. Daher möchte ich Ihnen hier die Grundzüge der maschinennahen Programmierung erläutern. Nach der Lektüre dieses Kapitels sind Sie in der Lage, die grundsätzliche Funktionsweise von Maschinenprogrammen zu verstehen und selbst zu entscheiden, ob Sie sich weiter mit dieser Sprache beschäftigen wollen. Sollte Ihnen die Maschinensprache nicht gefallen, so ist das auch kein Beinbruch. Das erworbene Wissen läßt sich auch für andere Aufgaben einsetzen, und schließlich sind PASCAL oder LOGO ja auch keine schlechten Wege, einem Computer etwas beizubringen.

13.1. Was ist Maschinensprache überhaupt?

Wie Sie sicher wissen, stellt die Maschinensprache die einzige Möglichkeit dar, den Prozessor ohne Umweg über einen Compiler oder Interpreter direkt zu programmieren. Daher ermöglicht diese Sprache auch so immens hohe Geschwindigkeiten.

Die Maschinensprache umfaßt verschiedene Befehle, aus denen sich alle komplexeren Operationen des BASICs oder anderer Sprachen zusammensetzen lassen. Man kann die Maschinenbefehle grob in drei Gruppen einteilen. Für BASIC-Programmierer am einfachsten zu verstehen sind die Sprungbefehle, mit denen das Programm ähnlich GOTO und GOSUB im Speicher umherspringen kann. Andere Befehle bewirken Datenmanipulationen (z.B. Additionen, Verknüpfungen etc.). Die letzte Gruppe umfaßt die Operationen, die Daten von einem Ort zum anderen innerhalb des Speichers bewegen.

Grundsätzlich gilt, daß es für Mikroprozessoren keine Variablen gibt. Sie kennen nur die normalen Speicherzellen und interne Register. Für die Unterscheidung zwischen Daten und Programmbytes muß der Programmierer selbst sorgen. Im allgemeinen können Datenmanipulationen nur in den internen Registern ablaufen.

Ein Maschinenbefehl besteht immer aus einem sogenannten Operationscode (oder Opcode), der sozusagen die "Nummer" des Befehls angibt. Dieser Opcode hat beim 6510 immer ein Byte. Außerdem können dem Befehl noch bis zu zwei Bytes an Daten folgen. Die 6510-Befehle sind also bis zu 3 Bytes lang.

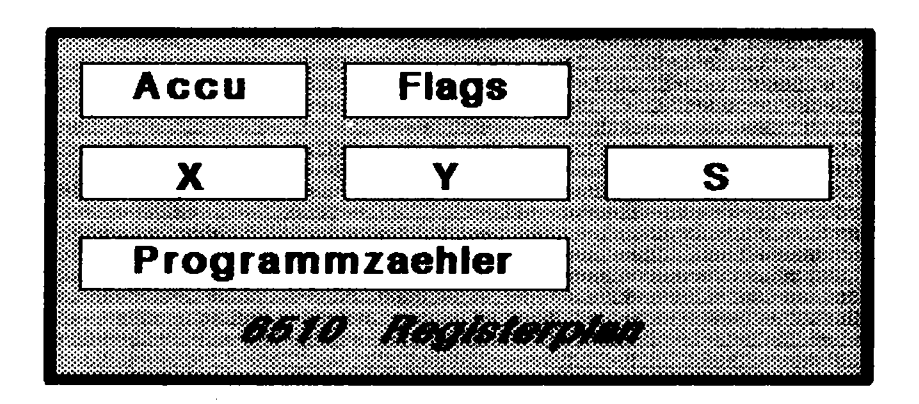
13.2. Der Takt

Alle Bauteile des Computers richten sich nach einem kleinen unscheinbaren Quarz, der den Takt (0,98 Megahertz = 980000 Schläge oder Zyklen pro Sekunde) vorgibt. Dies ist nötig, um die verschiedenen ICs zu synchronisieren. Geschähe dies nicht, so könnte es z.B. passieren, daß ein Speicherbaustein Daten zum Prozessor schickt, obwohl dieser noch gar nicht zur Übernahme bereit ist. Auch ein noch so schneller Mikroprozessor braucht immer noch ein wenig Zeit zur Verarbeitung der Daten.

13.3. Der Aufbau der Mikroprozessoren

Jeder Mikroprozessor besitzt interne Register, in denen die Operationen durchgeführt werden. Das wichtigste Register ist der sogenannte Akkumulator. In ihm laufen die meisten arithmetischen und logischen Verknüpfungen ab. Der Akkumulator (kurz Akku oder nur A) ist ein 8-Bit-Register, er kann also 1 Byte aufnehmen und bearbeiten (eigentlich bearbeitet der Akku selbst nichts, die Ergebnisse werden nur in diesem Register abgelegt). Die meisten Arithmetikbefehle brauchen zwei Operanden (z.B. die Addition von zwei Zahlen). Der erste Operand steht vor der Befehlsausführung schon im Akku, der zweite stammt aus einem anderen Register im Prozessor oder aus dem

Speicher. Nach der Addition wird das Ergebnis wieder im Akku gespeichert. Das ist bei allen Prozessoren gleich.



Ein anderes Register mit Namen P (Status des 6510) speichert verschiedene Flags, die bestimmte Zustände des Prozessors wiederspiegeln. Anhand dieser Flags kann zum Beispiel festgestellt werden, ob der Akkuinhalt 0 ist.

Der 6510 hat noch zwei Indexregister (X und Y) mit je 8 Bit. Diese lassen sich sehr vielseitig einsetzen, z.B. als Zählregister für Schleifen, um ganze Speicherseiten wie ein Array anzusprechen u.v.m.

13.4. Die Funktionsweise eines Mikroprozessors

Nehmen wir einmal an, im Speicher ihres Computers stehe ein Maschinenprogramm, das nur darauf wartet, ausgeführt zu werden. Natürlich muß sich der Mikroprozessor irgendwo merken, wo das Programm eigentlich steht. Dazu gibt es ein spezielles 16-Bit-Register, genannt Programmzähler (engl. Program Counter = PC). In ihm ist die Adresse des Befehls gespeichert, der als nächster zur Ausführung kommt. Soll der Befehl jetzt durchgeführt werden, so holt der Prozessor das Byte aus der angegebe-

nen Speicheradresse. Dieses Byte wird im Prozessor festgehalten und der Programmzähler um 1 erhöht, damit wir die Adresse des nächsten Bytes erhalten. Gleichzeitig wird der Opcode (um den handelt es sich nämlich bei dem Byte) dekodiert, d.h. der Mikroprozessor stellt fest, welcher der vielen Befehle da eigentlich im Speicher steht. Es kann vorkommen, daß noch ein oder zwei Bytes Daten folgen. Auch diese werden eingelesen, müssen jedoch nicht dekodiert werden. Sie gelangen statt dessen in bestimmte Register (womit der Befehl schon beendet sein könnte), oder werden zur Bearbeitung irgendwo im Prozessor bereitgehalten.

Müssen die Daten noch irgendwie verändert werden (z.B durch Addition o.ä.), so findet diese Operation jetzt statt und das Ergebnis wird wieder abgespeichert (z.B im Akku). Damit ist der Befehl beendet, der nächste kann gestartet werden.

Alle diese Vorgänge laufen natürlich nicht in nullkommanichts ab - auch Strom benötigt eben ein wenig Zeit zum Fließen. Der schnellste Befehl des 6510 braucht 2 Taktzyklen für seine Ausführung, andere brauchen z.B. 5.

Zusammenfassung:

Der PC zeigt immer auf die Speicherzelle, die das nächste zu bearbeitende Byte enthält. Nacheinander werden Opcode und Daten eingelesen. Der Opcode wird dekodiert, dann der Befehl ausgeführt.

13.5. Das Hexadezimalsystem

Wann immer Sie sich mit Maschinensprache beschäftigen, werden Sie auf die Zahlendarstellung im Hexadezimalsystem treffen. Dieses System besitzt im Gegensatz zu unserem herkömmlichen Dezimalsystem 16 Ziffern (0-9 und A-F für die Werte 10 bis 15). Es wird so häufig benutzt, weil die Umwandlung von Binär- in Hexzahlen sehr einfach ist. Ein weiterer Vorteil des Hexadezimalsystems ist es, daß eine Hexziffer genau ein halbes

Byte darstellt (die größte zweistellige Hexzahl FF entspricht der Binärkombination 1111 1111, dem größtmöglichen Inhalt eines Bytes). Man nimmt daher jeweils ein Halbbyte und wandelt es in eine Hexziffer um. Die Tabelle zeigt die dezimalen und binären Entsprechungen:

binaer	dez	hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Aus dem Byte $1010\ 1011_2$ wird also die hexadezimale Zahl AB_{16} (da $1010_2 = A_{16}$ und $1011_2 = B_{16}$). Natürlich funtkioniert das auch umgekehrt.

Für die Umwandlung von Hexzahlen in Dezimalzahlen werden zunächst alle Ziffern einzeln in das dezimale Äquivalent übersetzt. Die am weitesten rechts stehende Ziffer wird das mit 16^0=1, die zweite mit 16^1=16, die dritte mit 16^2=256 usw. multipliziert. Die erhaltenen Produkte werden dann addiert. Ein Beispiel:

```
ABCD16 ( entspricht 10,11,12,13)
= 10*16^3 + 11*16^2 + 12*16^1 + 13*16^0
= 10*4096 + 11*256  + 12*16   + 13*1
= 43981
```

Für den umgekehrten Weg (dez-hex) können Sie die Dezimalzahl durch 16 teilen und den entstehenden Divisionsrest als Hexziffer notieren. Das Ergebnis wird wieder durch 16 geteilt usw., bis es 0 wird. Auch hier ein Beispiel:

```
53000 / 16 = 3312 Rest 8  -) 8
 3312 / 16 =  207 Rest 0  -) 0
  207 / 16 =   12 Rest 15 -) F
   12 / 16 =    0 Rest 12 -) C
=) 53000 10 = CF08 16
```

Inzwischen gibt es Taschenrechner, die eine spezielle Funktion für die Basisumwandlung besitzen. Gute Assembler bzw. Hexmonitore bieten diese Funktion ebenfalls.

13.6. Binärarithmetik

13.6.1. Addition

Um es gleich zu Anfang zu sagen: Die binäre Addition unterscheidet sich von der dezimalen nur im Zahlensystem, ansonsten funktioniert sie genauso.

Die Summen von zwei Nullen oder einer Null und einer Eins (egal in welcher Reihenfolge) bedürfen keiner Erläuterung, hier wird ganz normal addiert. Wollen wir jedoch 1 + 1 rechnen, so ergibt sich ein Problem. In der dezimalen Entsprechung wäre das Ergebnis eine 2. Die gibt es jedoch im binären System nicht. Also muß (wie beim Überschreiten der 9 im Dezimalsystem) ein Übertrag auf die nächste Stelle gemacht werden:

```
  0     0     1     1
+ 0   + 1   + 0   + 1
...   ...   ...   ...
  0     1     1    10
```

Auch ganze Bytes lassen sich sehr einfach verknüpfen. Hier wird einfach jede Stelle für sich addiert (und ein eventueller Übertrag beachtet):

```
  01101101 =  109
+ 00001001 = + 9

     1  1    (Überträge)
..........   ....
  01110110 =  118
```

Zur besseren Übersicht sind hier die Überträge aufgeführt worden.

Sollte es vorkommen, daß zwei Einsen addiert werden müssen und noch ein Übertrag dazukommt (1+1+1=3) so ist das Ergebnis 1 1 (eigentlich klar!)

Versuchen Sie einmal diese Addition:

```
   10010011
 + 11011111

  111111    (Überträge)
-------------
  101110010
```

Jetzt haben wir im Ergebnis plötzlich 9 Bits! Das neunte Bit heißt Carry- oder Übertrags-Bit. Es zeigt an, daß die Addition von zwei 8-Bit-Zahlen den zulässigen Bereich für ein Byte (0 - 255) überschritten hat, womit wir auch schon bei der 16-Bit-Addition sind. Kein Computer kommt mit nur 8 Bits für die Zahlendarstellung aus, die Zahlen haben meist einen viel größeren Bereich. Tatsache ist aber, daß ein 8-Bit-Mikroprozessor immer nur 8 Bits gleichzeitig verarbeiten kann. Besteht eine Zahl z.B. aus zwei Bytes, so muß die Addition nacheinander an beiden durchgeführt werden. Da bis auf den Übertrag die beiden Teile der Zahl völlig unabhängig voneinander addiert werden können, braucht man nur das Carry-Bit, um auch größere Zahlen zu bearbeiten. Es hat die Aufgabe, den Übertrag von der letzten Stelle des ersten Bytes zur ersten Stelle des zweiten Bytes zwischenzuspeichern.

Ein Beispiel:

```
   00110101 10010011
 + 10011011 11011111

    1111111   11111  (Überträge)
------------------
   11010001 01110010
```

Den rechten Teil der Addition kennen Sie bereits aus dem vorherigen Beispiel.

13.6.2. Subtraktion

Wenn ein Computer eine Zahl von einer anderen subtrahieren will, so bildet er zunächst das negative Äquivalent dieser Zahl (d.h. er multipliziert mit -1) und addiert es dann. Dies läuft so ab, weil eine Addition und eine Negierung aus elektronischen Grundbausteinen (wie AND, OR, XOR, NOT) zusammengesetzt werden kann, nicht aber eine Subtraktion.

Um eine negative Zahl darzustellen, wird der Zahlenbereich eines Bytes von 0 - 255 nach -127 bis +127 verschoben. Das höchstwertige Bit (Bit 7) dient dann als Vorzeichen. Ist es auf 1, so haben wir eine negative Zahl vor uns, bei 0 ist das Byte positiv. Dabei kann aber zur Negierung einer Zahl nicht einfach Bit 7 gesetzt werden. Ein Beispiel verdeutlicht die Schwierigkeiten:

```
  00000001
+ 10000001
----------
  10000010
```

In Dezimalsystem übertragen würde dies bedeuten, daß 1 + (-1) = -2 ist. Deshalb wird ein anderer Weg gegangen. Ein Byte kann durch Bildung des sogenannten Zweierkomplements sehr einfach mit -1 multipliziert werden. Dazu werden alle Bits invertiert und zusätzlich eine 1 addiert.

```
Beispiel:    01011011
invertiert:  10100100
           +        1
           ----------
             10100101
```

Wenn wir nach diesem Schema 1 - 1 im binären System berechnen, so erhalten wir das richtige Ergebnis:

```
  00000001
+ 11111111

 11111111  (Überträge)
----------
 100000000
```

Wie Sie sehen, entsteht scheinbar ein Übertrag. Doch auch hier verhält sich die Subtraktion anders. Wir können es hier einfach ignorieren. Würden wir 16 Bits subtrahieren, so würde unser jetzt überflüssiges Carry-Bit dafür sorgen, daß die Stellen des zweiten Bytes auch auf 0 gesetzt würden. Das ist wichtig, da bei negativen 16-Bit-Zahlen alle 16 Stellen invertiert werden. Als Zwei-Byte-Zahl sähe -1 also so aus: 11111111 11111111. Fehlte das Carry-Bit jetzt, so lautete unser Ergebnis 11111111 00000000. Und das ist falsch!

Zum Glück ist die Programmierung einer Subtraktion nicht so kompliziert. Die Subtraktionsbefehle beider Mikroprozessoren beinhalten bereits die Bildung des Zweierkomplements

13.6.3. Multiplikation

Auch wenn Sie es nicht glauben: Die Maschinensprache hat nur zwei Rechenbefehle, und zwar für Addition und Subtraktion. Alle anderen Rechenarten werden aus diesen Grundbefehlen zusammengesetzt, meist als Unterprogramm.

Da wir nicht in allen Einzelheiten in die Maschinensprache einsteigen wollen (dazu gibt es bessere und ausführlichere Literatur), stelle ich Ihnen nur den einfachsten Algorithmus zur Mul-

tiplikation vor. Er wird von Profis nicht gern benutzt, da er nicht sehr effizient ist. Nun aber zur Sache.

Um das Produkt x * n zu berechnen, genügt es, x n-mal zu addieren. Dies funktioniert natürlich nur bei ganzen Zahlen. Für Dezimalbrüche gibt es kompliziertere Verfahren, bei denen Zahlen z.B. Stelle für Stelle und nicht als Ganzes miteinander verknüpft werden, die aber im Prinzip ähnlich funktionieren.
Zum besseren Verständnis noch ein Beispiel:

```
4 * 3 = 4 + 4 + 4 = 12
```

13.6.4. Division

Auch für die Division gibt es ein sehr einfaches Verfahren. Um x durch n zu teilen, wird einfach fortwährend n von x abgezogen. Die Anzahl der möglichen Subtraktionen, bis n größer x wird, gibt das Ergebnis der Division an. Hier ein Beispiel:

```
10 / 3 = ?
10 - 3 = 7    Zählregister= 1
 7 - 3 = 4    Zählregister= 2
 4 - 3 = 1    Zählregister= 3
=) 10 / 3 = 3 Rest 1
```

Diese Methoden sind möglich, weil die Maschinensprache so ungeheuer große Geschwindigkeiten erlaubt. Übrigens arbeitet auch ein Taschenrechner nach diesem Prinzip. Jedesmal, wenn Sie eine Rechentaste drücken, läuft ein kleines Maschinenprogramm (natürlich mit den erwähnten aufwendigen Algorithmen) ab.

Aus den 4 Grundrechenarten lassen sich dann noch höhere Funktionen (z.B. Potenzen, Sinus o.ä.) zusammensetzen. Auf diese Art und Weise kann jede mathematische Operation durch

kleinste AND-, OR-, XOR- und NOT-Operationen ausgedrückt werden (da Addition und Subtraktion sich aus letzteren konstruieren lassen).

13.7. Wie funktionieren Vergleiche?

Im BASIC stellen Vergleiche nichts Ungewöhnliches dar. Doch wie kann man sie in Maschinensprache erzeugen? Sehen wird uns dazu einmal ein Beispiel an:

```
A = B   (=)   A - B = 0
```

Wie Sie sehen, kann ein Vergleich zwischen 2 Zahlen (hier A und B) recht einfach umgeformt werden. Für den Computer hat diese Form den Vorteil, daß auf der rechten Seite der Gleichung eine 0 steht. Die 0 ist die einzige Zahl, von der der Mikroprozessor feststellen kann, ob sie gerade im internen Rechenregister (meist Accu genannt) steht oder nicht. Dazu werden einfach alle Bits miteinander ODER-verknüpft - etwa so:

```
Bit 7 OR Bit 6 OR Bit 5 OR Bit 4 OR Bit 3 OR Bit 2 OR Bit 1 OR Bit 0
```

Wenn alle 8 Bits des Accus auf 0 waren, so ist das Ergebnis dieser Verknüpfungskette eine 0, in allen anderen Fällen (d.h. wenn mindestens ein Bit auf 1 ist) ist das Ergebnis 1. So kann der Mikroprozessor angeben, ob das Rechenregister (wo fast immer das Ergebnis der letzten Operation steht) gleich oder ungleich 0 ist - voila, die ersten beiden Vergleiche sind erzeugt. Für einen Vergleich A=B oder A ungleich B brauchen wir also nur die beiden Zahlen voneinander zu subtrahieren und dann festzustellen, ob der Inhalt des Accus 0 ist. Dies können Sie mittels des Z-Flags (Z steht für Zero). Ist es auf 1, so ist das Ergebnis der letzten Operation 0 gewesen. Ist Z=0, so war das letzte Ergebnis ungleich 0. Das Flag ist also nicht nur auf den Akku beschränkt, andererseits verändern aber auch nicht alle

Befehle die Flags. Ob das geschieht, können Sie in der Befehlsliste im Anhang nachlesen. Doch nun zurück zu den Vergleichen.

Bei "größer" und "kleiner" gehen wir fast wie bei "gleich" vor. Nach der Subtraktion sehen wir nach, ob die Zahl im Accu kleiner oder größer 0 ist, erkennbar am Vorzeichenbit:

```
A größer  B   (=)   A - B größer 0  (erfüllt, wenn Bit 7 = 0)
A kleiner B   (=)   A - B kleiner 0 (erfüllt, wenn Bit 7 = 1)
```

Das Vorzeichenbit wird von vielen Befehlen in das N(egative)-Flag übertragen. Dort kann es mittels spezieller Befehle einfach abgefragt werden.

Ein weiteres Flag heißt V. Es meldet nach einigen arithmetischen Operationen, ob das Vorzeichenbit fehlerhaft verändert wurde (das braucht uns hier nicht zu interessieren; wir möchten ja nur in die Maschinensprache hineinriechen).

13.8. 6510-Maschinensprache

Jetzt geht es endlich zur Sache. Bevor wir jedoch anfangen, unsere ersten Schritte in der schnellsten Programmiersprache der Welt zu machen, sollten Sie ein bis zwei Blicke in den Anhang werfen.Dort finden Sie Beschreibungen aller Befehle des 6510, von denen Sie sich schon einmal einen kleinen Überblick verschaffen sollten. Keine Angst - Sie brauchen nicht alles zu verstehen.

Wir werden die ersten Gehversuche in Maschinensprache mit Additionen und Subtraktionen machen, weil Sie dadurch die Arbeitsweise eines Mikroprozessors am besten kennenlernen. Beginnen wir deshalb mit einem Additionsprogramm für zwei 8-Bit-Zahlen.

Die beiden Bytes, die addiert werden sollen, legen wir vorher per POKE in zwei Speicherzellen ab. Dies ist zwar keine besonders komfortable Methode, doch sie reicht aus. Einen INPUT-Befehl gibt es in der Maschinensprache leider nicht.

Der erste Grundsatz der Assemblerprogrammierung lautet: (fast) alle Datenmanipulationen laufen im Accu ab. Daher muß der Accu durch den ersten Befehl im Programm mit der ersten Zahl geladen werden. Dazu dient LDA $nnnn. Da es sein kann, daß das Carrybit noch gesetzt ist, müssen wir es mit CLC löschen. Dann kommt der eigentliche Additionsbefehl ADC $mmmm. Dieser Befehl holt sich die zweite Zahl aus der Speicherzelle mmmm ab und addiert sie zum Accuinhalt. Das Ergebnis dieser Addition steht dann wieder im Accu.

Da dieses Ergebnis angezeigt werden soll, befördert es der Befehl STA $xxxx in eine Speicherzelle, von wo es gePEEKd wird. Damit ist die Addition beendet. Es fehlt nur noch der Rücksprung zum BASIC durch RTS.

Das ganze Programm sieht also jetzt so aus:

```
LDA $nnnn
CLC
ADC $mmmm
STA $xxxx
RTS
```

Diese Befehle sollen irgendwo im Speicher abgelegt werden. Zuvor müssen wir uns aber noch Gedanken um den Ort der Speicherung machen. Zum Glück haben die Commodore-Entwickler einen speziellen Bereich für Maschinenspracheprogramme reserviert. Er liegt bei C000 bis CFFF. Am besten wird das Programm ab der Adresse C000 abgelegt, die Datenregister kommen an das Ende dieses Bereiches.

Um das Programm einzugeben, müssen wir uns allerdings noch etwas Mühe machen. Wir müssen nämlich zu jedem Befehl den

Opcode aus dem Anhang heraussuchen (Schriftsprache versteht der 6510 leider noch nicht).

Damit das ganze nicht zu sehr in Arbeit ausartet, habe ich die entsprechenden Werte schon einmal herausgesucht:

```
Adr.  Codes     Befehl
-----------------------
C000  AD FD CF LDA $CFFD
C003  18       CLC
C004  6D FE CF ADC $CFFE
C007  8D FF CF STA $CFFF
C00A  60       RTS
```

Bitte beachten Sie, daß bei den Adressen das Lowbyte immer vor dem Highbyte stehen muß!

Die Codes haben wir, aber damit ist das Programm noch nicht im Speicher. Dazu brauchen wir ein Ladeprogramm wie dieses:

```
10 FOR I= 49152 TO 49162
20 READ A: POKE I,A: NEXT
30 DATA 173, 253, 207
40 DATA  24
50 DATA 109, 254, 207
60 DATA 141, 255, 207
70 DATA  96
```

Das Programm ist also jetzt im Speicher, wir müssen nun nur noch die zu addierenden Zahlen eingeben. Auch dazu bietet das BASIC eine Möglichkeit. Wählen Sie sich zwei Zahlen im Bereich von 0 bis 255 und POKEn Sie diese in die Speicherzellen 53245 und 53246. Schon kann es losgehen.

Das Maschinenprogramm kann jetzt mit SYS 49152 gestartet werden. Sollte danach der Cursor nicht sofort wieder erscheinen,

haben Sie irgendetwas falsch gemacht. In diesem Fall müssen Sie den Rechner ausschalten und neu anfangen. Sonst können Sie das Ergebnis der Addition durch PRINT PEEK (53247) erfahren.

Es empfiehlt sich, dieses Programm mehrmals mit verschiedenen Werten zu starten, um die Addition richtig durchzuprobieren und zu verstehen.

13.9. Der zweite Schritt: 16-Bit-Addition

Wie schon erwähnt, braucht man zur Behandlung größerer Zahlen die 16-Bit-Addition oder noch aufwendigere Algorithmen. Unten wird ein Programm beschrieben, daß eine beliebige 16-Bit-Zahl zu einer Konstanten addiert. Die Zahl wird in die Speicherzellen CFFF (Highbyte) und CFFE (Lowbyte) gebracht. Da die Zahl 16 Bits hat, werden zwei Bytes und zwei Additionsschritte gebraucht. Zunächst beginnt jedoch alles normal mit

```
LDA $CFFE
```

und

```
CLC.
```

Da jedoch eine Konstante addiert werden soll, benutzen wir jetzt ADC #Lowbyte. Dieser Befehl holt sich die zweite Zahl jetzt nicht mehr aus einer angegebenen Adresse ab, sondern nimmt direkt das nachfolgende Byte (also die gewünschte Konstante).

Nach diesem Befehl ist die niederwertige Hälfte des Ergebnisses bereits komplett. Sie wird durch STA $CFFC "gerettet". Ein eventueller Übertrag ist jetzt im Carrybit gespeichert und wird auch durch das Laden der zweiten Hälfte per LDA $CFFF nicht gelöscht. Jetzt kann wieder normal addiert werden mit ADC #Highbyte.

STA $CFFD bringt auch den zweiten Teil des Ergebnisses in den Speicher. Mit RTS wird das Programm abgeschlossen. Hier das ganze Listing:

```
C000 LDA $CFFE
C003 CLC
C004 ADC #E8
C006 STA $CFFC
C009 LDA $CFFF
C00C ADC #03
C00E STA $CFFD
C011 RTS
```

Als Konstante wurde 03E8 (= 1000 dezimal) gewählt.

Hier wieder das Ladeprogramm:

```
10 FOR I = 49152 TO 49169
20 READ A: POKE I,A: NEXT
30 DATA 173, 254, 207
40 DATA  24
50 DATA 105, 232
60 DATA 109, 252, 207
70 DATA 173, 255, 207
80 DATA 105,   3
90 DATA 109, 253, 207
100 DATA  96
```

13.10. Subtraktion

Da die Subtraktion bis auf Carry-Bit und Rechenbefehl der Addition entspricht, soll hier nur kurz das Programm aufgelistet werden (vgl. 8-Bit-Addition).

```
C000 LDA $CFFF
C003 SEC          (Carry setzen)
C004 SBC $CFFE    (subtrahieren)
C007 STA $CFFD
C00A RTS
```

13.11. Multiplikation

Wie Sie aus Kapitel 13.6. wissen, stellt eine Multiplikation eine mehrfache Addition dar. Dies wollen wir uns jetzt zunutze machen, um eine Multiplikation und so ganz nebenbei auch ein Unterprogramm in Maschinensprache zu erstellen.

Bei der beschriebenen Methode liegt es nahe, die Addition in ein eigenständiges Unterprogramm zu verlegen (was zwar eigentlich nicht nötig wäre, aber so stillen wir wenigstens unseren Bildungshunger) und dieses von einer Schleife aus aufzurufen. Beginnen wir wieder bei der Addition.

Bei der Multiplikation von zwei 8-Bit-Zahlen erhalten wir ein 16-Bit-Ergebnis. Die Additionsroutine muß daher ein einzelnes Byte zu einer 2-Bytezahl addieren. Dies läßt sich vereinfachen, wenn man sich zur 8-Bit-Zahl eine 0 als Highbyte dazudenkt und dann eine normale 16-Bit-Addition durchführt. Damit wird der Übertrag zum Highbyte des Ergebnisses gewährleistet. Das sieht dann so aus:

```
LDA $CFFE    (Lowbyte des Ergebnisses)
CLC
ADC $CFFC    (8-Bit-Zahl addieren)
STA $CFFE    (zurückspeichern)
LDA $CFFF    (Highbyte des Ergebnisses)
ADC #00      (0 und Carry-Bit addieren)
STA $CFFF    (zurückspeichern)
RTS          (Unterprogrammende)
```

Nach der Addition steht das Ergebnis in CFFE/CFFF, die 8-Bit-Zahl stand vorher in CFFC. Jetzt fehlt nur noch die Schleife. Die Länge wird durch den Multiplikator vorgegeben, der vorher in Speicherzelle CFFD abgelegt wurde.

Die einfachste Methode, eine Schleife mit variabler Länge zu programmieren, besteht darin, nach jedem Durchlauf ein spezielles Register um 1 zu vermindern. Ist das Register auf 0 heruntergezählt, so kann die Schleife beendet werden. Dazu eignet sich am besten das X-Register. Am Beginn der Schleife wird durch LDX $CFFD der Zähler initialisiert (CFFD enthält ja den Multiplikator).

Dann folgt der Unterprogrammaufruf mit JSR $Addition (für Addition wird die Einsprungadresse eingesetzt). Nach dem Unterprogramm muß der Schleifenzähler dekrementiert (d.h. um 1 vermindert) werden. Dies übernimmt der Einbytebefehl DEX. Das besondere an ihm ist, daß auch er die Z- und N-Bits verändert. Damit wird dann angezeigt, ob der Inhalt des X-Registers Null oder negativ ist. Die Steuerbits beziehen sich also nicht nur auf den Accu.

Die Opcode-Tabelle im Anhang zeigt, welche Befehle welche Bits verändern können. Anhand der Steuerbits können die sogenannten Branch-Befehle bedingte Verzweigungen ausführen. Solch eine Bedingung finden wir auch in unserer Schleife, sie soll ja bei X=0 abgebrochen werden. Ist X ungleich 0, so soll ein Rücksprung erfolgen. Dafür ist BNE $Adresse (branch on not equal to zero) zuständig. Ist das Zerobit auf 1, so heißt dies, daß die letzte Operation das Ergebnis 0 hatte, bei Z=0 war es ungleich 0. Der BNE-Befehl prüft das Z-Bit. Ist es auf 0, so verzweigt er zur angegebenen Adresse, sonst wird mit dem nächsten Befehl weitergemacht. Der Branch-Befehl arbeitet mit der sogenannten relativen Adressierung, d.h. es wird der ABSTAND zum Sprungziel angegeben. Dieser Abstand ist allerdings leicht zu berechnen. Nehmen Sie den auf den BRANCH-Befehl folgenden Befehl und errechnen Sie dessen Abstand zum Sprungziel. Hat das Ziel eine kleinere Adresse, so wird das Ergebnis negativ. Diese Zahl wandeln Sie jetzt in eine 8-Bit-Zahl (mit Vorzeichen) - fertig. Da der besagte Abstand nur ein

Byte umfassen darf, sind leider nur Sprünge um maximal 129 Bytes nach vorne und um 126 Bytes zurück möglich. Das ist zwar auf den ersten Blick wenig, ist aber in der Praxis nur selten störend.

Bevor die Schleife beginnt, müssen die beiden Ergebnisbytes allerdings noch gelöscht werden. Sehen Sie es sich selbst an:

```
C000 LDA #00
C002 STA $CFFF    (17FF löschen)
C005 STA $CFFE    (17FE löschen)
C008 LDX $CFFD    (X laden)
C00B JSR $C00E    (Unterprogrammaufruf)
C00E DEX          (X dekrementieren)
C00F BNE $FA      (Verzweigung -6 zu C00B)
C011 RTS          (Hauptprogrammende)
C012 LDA $CFFE    (Additionsupg.)
C015 CLC
C016 ADC $CFFC
C019 STA $CFFE
C01C LDA $CFFF
C01F ADC #00
C021 STA $CFFF
C024 RTS          (Upg.-Ende)
```

Jetzt sollten Sie das Programm eigentlich selbst codieren können.

Anhang

I. Maschinenbefehle 6510

ADC
Die dem Befehl folgenden Daten und das Carrybit werden zum Akku addiert. Danach werden die Flags N, V, Z und C(arry) dem Ergebnis entsprechend gesetzt.

AND
Die dem Befehl folgenden Daten werden mit dem Akku UND-verknüpft, das Ergebnis wieder im Akku abgelegt. Die Flags N und Z werden aktualisiert.

ASL
Die Bits des Akkus oder einer Speicherzelle werden um eine Stelle nach links geschoben. Das Bit 7 gelangt ins Carrybit, Bit 0 wird mit 0 "aufgefüllt". Die Flags N, Z, und C werden verändert.

BCC dd
Wenn das Carrybit auf 0 ist, springt der Prozessor zur Adresse, die sich aus der Adresse des nächsten Befehls + dd ergibt. Flags werden nicht verändert.

BCS dd
Wie BCC, springt jedoch bei Carry=1

BEQ dd
Wie BCC, springt jedoch bei Z=1

BIT
Die Bits einer Speicherzelle werden mit dem Akku UND-verknüpft. Das Ergebnis wird nicht gespeichert; war es ungleich 0, so wird Z=1, sonst bleibt Z=0.

So können Bits einer Speicherzelle getestet werden (dazu muß nur das entsprechende Bit im Akku auf 1 gebracht werden).

Außerdem werden die beiden höchstwertigen Bits aus dem Speicher in die Flags N (Bit 7) und V (Bit 6) übertragen.

BMI dd
Wie BCC, springt jedoch bei N=1

BNE dd
Wie BCC, springt jedoch bei Z=0

BPL dd
Wie BCC, springt jedoch bei N=0

BRK
Löst einen künstlichen Interrupt aus. Zur Unterscheidung wird jedoch Flag B auf 1 gesetzt. Wird üblicherweise zum Testen von Programmen benutzt.

BVC dd
Wie BCC, springt jedoch bei V=0

BVS dd
Wie BCC, springt jedoch bei V=1

CLC
Löscht das Carrybit

CLD
Löscht das D-Flag. Folge: BCD-Modus wird abgeschaltet.

CLI
Löscht das Interrupt-Flag. Folge: Interrupts werden erlaubt.

CLV
Löscht das V-Flag.

CMP
Die nachfolgenden Daten werden mit dem Akku verglichen (subtrahiert). Wenn beide Bits gleich sind, wird Z=1. Ist der Inhalt des Akkus kleiner als die Daten, so ist N=1. Das Carrybit wird auf 1 gesetzt, wenn der Akku größer oder gleich ist. Mit den entsprechenden Branch-Befehlen kann das Ergebnis ausgewertet werden.

CPX
Wie CMP, die Daten werden jedoch mit dem X-Register verglichen.

CPY
Wie CMP, die Daten werden jedoch mit dem Y-Register verglichen.

DEC
Die angegebene Speicherzelle wird dekrementiert (der Inhalt um 1 vermindert), die Flags N und Z verändert.

DEX
Wie DEC, jedoch wird das X-Register dekrementiert.

DEY
Wie DEC, jedoch wird das Y-Register dekrementiert.

EOR
Die dem Befehl folgenden Daten werden mit dem Akku XOR-verknüpft, das Ergebnis im Akku abgelegt. Die Flags N und Z werden aktualisiert.

INC
Die angegebene Speicherzelle wird inkrementiert (der Inhalt um 1 erhöht), die Flags N und Z verändert.

INX
Wie INC, jedoch wird das X-Register dekrementiert.

INY
Wie INC, jedoch wird das Y-Register dekrementiert.

JMP
Der Prozessor springt zur angegebenen Adresse.

JSR
Der Prozessor springt zum Unterprogramm an der angegebenen Adresse.

LDA
Der Akkumulator wird mit den dem Befehl folgenden Daten geladen. Dabei werden die Flags N und Z aktualisiert.

LDX
Wie LDA, jedoch wird in das X-Register geladen.

LDY
Wie LDA, jedoch wird in das Y-Register geladen.

LSR
Die Bits des Akkus oder einer Speicherzelle werden um eine Stelle nach rechts geschoben. Das Bit 0 gelangt ins Carrybit, Bit 7 wird mit 0 "aufgefüllt". Die Flags N, Z, und C werden verändert.

NOP
Wartet zwei Taktzyklen lang.

ORA
Die dem Befehl folgenden Daten werden mit dem Akku ODER-verknüpft, das Ergebnis wieder im Akku abgelegt. Die Flags N und Z werden aktualisiert.

PHA
Der Akku wird auf den Stapel gebracht.

PHP
Die Flags werden auf den Stapel gebracht.

PLA
Ein Byte wird vom Stapel in den Akku geholt. Die Flags N und Z werden aktualisiert.

PLP
Die Flags werden vom Stapel geholt.

ROL
Die Bits des Akkus oder einer Speicherzelle werden um eine Stelle nach links rotiert. Bit 7 gelangt ins Carrybit, dessen alter Inhalt in Bit 0 geschoben wird. Die Flags N, Z und C werden verändert.

ROR
Die Bits des Akkus oder einer Speicherzelle werden um eine Stelle nach rechts rotiert. Bit 0 gelangt ins Carrybit, dessen alter Inhalt in Bit 7 geschoben wird. Die Flags N, Z und C werden verändert.

RTI
Rücksprung aus Interruptroutine, stellt alten Stand des Programmzählers und der Flags wieder her.

RTS
Rücksprung aus Unterprogramm, stellt den alten Stand des PCs wieder her.

SBC
Wie ADC, es wird jedoch subtrahiert.

SEC
Setzt das Carrybit

SED
Löscht das D-Flag. Folge: BCD-Modus wird eingeschaltet.

SEI
Löscht das Interrupt-Flag. Folge: Interrupts werden ignoriert.

STA
Speichert den Inhalt des Akkus an der angegebenen Adresse ab.

STX
Speichert den Inhalt des X-Registers an der angegebenen Adresse ab.

STY
Speichert den Inhalt des Y-Registers an der angegebenen Adresse ab.

TAX
Der Inhalt des Akkus wird in das X-Register geschrieben. Die Flags N und Z werden aktualisiert.

TAY
Der Inhalt des Akkus wird in das Y-Register geschrieben. Die Flags N und Z werden aktualisiert.

TSX
Der Stapelzeiger wird ins X-Register kopiert. Die Flags N und Z werden verändert.

TXA
Der Inhalt des X-Registers wird in den Akku geschrieben. Die Flags N und Z werden verändert.

TXS
Der Inhalt des X-Registers wird in den Stapelzeiger geschrieben.

TYA
Der Inhalt des Y-Registers wird in den Akku geschrieben. Die Flags N und Z werden verändert.

II. Opcodeliste 6510

Befehl	Opcode	Flags	Beschreibung
ADC $nnnn	6Dnnnn	NV ZC	add. Speicherzelle nnnn zum Akku
ADC $nn	65nn	NV ZC	addiert Speicherzelle nn
ADC #$nn	69nn	NV ZC	addiert Byte nn zum Akku
ADC $nnnn,X	7Dnnnn	NV ZC	add. Speicherzelle nnnn+X
ADC $nnnn,Y	79nnnn	NV ZC	add. Speicherzelle nnnn+X
ADC $nn,X	75nn	NV ZC	add. Speicherzelle nn+X
ADC ($nn,X)	61nn	NV ZC	add. Byte aus (nn+X)
ADC ($nn),Y	71nn	NV ZC	add. Byte aus (nn)+Y
AND $nnnn	2Dnnnn	N Z	UND-Verkn. Zelle nnnn mit Akku
AND $nn	25nn	N Z	UNDed Speicherzelle nn
AND #$nn	29nn	N Z	UNDed Byte nn mit Akku
AND $nnnn,X	3Dnnnn	N Z	UNDed Speicherzelle nnnn+X
AND $nnnn,Y	39nnnn	N Z	UNDed Speicherzelle nnnn+X
AND $nn,X	35nn	N Z	UNDed Speicherzelle nn+X
AND ($nn,X)	21nn	N Z	UNDed Byte aus (nn+X)
AND ($nn),Y	31nn	N Z	UNDed Byte aus (nn)+Y
ASL A	0A	N ZC	arithm. Linksschieben d. Akkus
ASL $nnnn	0Ennnn	N ZC	der Speicherzelle nnnn
ASL $nn	06nn	N ZC	der Speicherzelle nn
ASL $nnnn,X	1Ennnn	N ZC	der Speicherzelle nnnn+X
ASL $nn,X	16nn	N ZC	der Speicherzelle nn+X
BCC $dd	90dd		verzweigt zu PC+dd, wenn C=0
BCS $dd	B0dd		verzweigt zu PC+dd, wenn C=1
BEQ $dd	F0dd		verzweigt zu PC+dd, wenn Z=1
BIT $nnnn	20nnnn	NV Z	testet Zelle nnnn
BIT $nn	24nn	NV Z	testet Zelle nn
BMI $dd	30dd		verzweigt zu PC+dd, wenn N=1
BNE $dd	D0dd		verzweigt zu PC+dd, wenn Z=0
BPL $dd	10dd		verzweigt zu PC+dd, wenn N=0
BRK	00	B I=1	Software-Interrupt
BVC $dd	50dd		verzweigt zu PC+dd, wenn V=0
BVS $dd	70dd		verzweigt zu PC+dd, wenn V=1
CLC	18	C=0	löscht C
CLD	D8	D=0	löscht D

```
CLI          58       I=0   löscht I
CLV          B8      V=0    löscht V
CMP $nnnn    CDnnnn N    ZC vergleicht Zelle nnnn mit Akku
CMP $nn      C5nn   N    ZC vergl. Speicherzelle nn
CMP #$nn     C9nn   N    ZC vergl. Byte nn mit Akku
CMP $nnnn,X  DDnnnn N    ZC vergl. Speicherzelle nnnn+X
CMP $nnnn,Y  D9nnnn N    ZC vergl. Speicherzelle nnnn+X
CMP $nn,X    D5nn   N    ZC vergl. Speicherzelle nn+X
CMP ($nn,X)  C1nn   N    ZC vergl. Byte aus (nn+X)
CMP ($nn),Y  D1nn   N    ZC vergl. Byte aus (nn)+Y
CPX $nnnn    ECnnnn N    ZC vergleicht Zelle nnnn mit X
CPX $nn      E$nn   N    ZC vergl. Speicherzelle nn
CPX #$nn     E0nn   N    ZC vergl. Byte nn mit X
CPY $nnnn    CCnnnn N    ZC vergleicht Zelle nnnn mit Y
CPY $nn      C4nn   N    ZC vergl. Speicherzelle nn
CPY #$nn     C0nn   N    ZC vergl. Byte nn mit Y
DEC $nnnn    CEnnnn N    Z  dekrementiert Speicherzelle nnnn
DEC $nn      C6nn   N    Z  dekremnetiert Speicherzelle nn
DEC $nnnn,X  DEnnnn N    Z  dekrement. Speicherzelle nnnn+X
DEC $nn,X    D6nn   N    Z  dekrementiert Speicherzelle nn+X
DEX          CA     N    Z  dekrementiert X
DEY          88     N    Z  dekrementiert Y
EOR $nnnn    4Dnnnn N    Z  XOR-verkn. Zelle nnnn mit Akku
EOR $nn      45nn   N    Z  XORed Speicherzelle nn
EOR #$nn     49nn   N    Z  XORed Byte nn mit Akku
EOR $nnnn,X  5Dnnnn N    Z  XORed Speicherzelle nnnn+X
EOR $nnnn,Y  59nnnn N    Z  XORed Speicherzelle nnnn+X
EOR $nn,X    55nn   N    Z  XORed Speicherzelle nn+X
EOR ($nn,X)  41nn   N    Z  XORed Byte aus (nn+X)
EOR ($nn),Y  51nn   N    Z  XORed Byte aus (nn)+Y
INC $nnnn    EEnnnn N    Z  inkrementiert Speicherzelle nnnn
INC $nn      E6nn   N    Z  inkremnetiert Speicherzelle nn
INC $nnnn,X  FEnnnn N    Z  inkrement. Speicherzelle nnnn+X
INC $nn,X    F6nn   N    Z  inkrementiert Speicherzelle nn+X
INX          EA     N    Z  inkrementiert X
INY          C8     N    Z  inkrementiert Y
JMP $nnnn    4cnnnn         springt nach nnnn
JMP $(nnnn)  6cnnnn         springt nach (nnnn)
JSR $nnnn    20             springt zum Upg. nnnn
LDA $nnnn    ADnnnn N    Z  lädt Speicherzelle nnnn in Akku
```

```
LDA $nn     A5nn   N    Z  lädt Speicherzelle nn
LDA #$nn    A9nn   N    Z  lädt Byte nn in Akku
LDA $nnnn,X BDnnnn N    Z  lädt Speicherzelle nnnn+X
LDA $nnnn,Y B9nnnn N    Z  lädt Speicherzelle nnnn+Y
LDA $nn,X   B5nn   N    Z  lädt Speicherzelle nn+X
LDA ($nn,X) A1nn   N    Z  lädt Byte aus (nn+X)
LDA ($nn),Y B1nn   N    Z  lädt Byte aus (nn)+Y
LDX $nnnn   AEnnnn N    Z  lädt Speicherzelle nnnn in X
LDX $nn     A6nn   N    Z  lädt Speicherzelle nn
LDX #$nn    A2nn   N    Z  lädt Byte nn in X
LDX $nnnn,Y BEnnnn N    Z  lädt Speicherzelle nnnn+Y
LDX $nn,Y   B6nn   N    Z  lädt Byte aus (nn)+Y
LDY $nnnn   ACnnnn N    Z  lädt Speicherzelle nnnn in Y
LDY $nn     A4nn   N    Z  lädt Speicherzelle nn
LDY #$nn    A0nn   N    Z  lädt Byte nn in Y
LDY $nnnn,X BCnnnn N    Z  lädt Speicherzelle nnnn+X
LDY $nn,X   B4nn   N    Z  lädt Byte aus (nn)+X
LSR A       4A     N=0  ZC log. Rechtsschieben d. Akkus
LSR $nnnn   4Ennnn N=0  ZC   der Speicherzelle nnnn
LSR $nn     46nn   N=0  ZC   der Speicherzelle nn
LSR $nnnn,X 5Ennnn N=0  ZC   der Speicherzelle nnnn+X
LSR $nn,X   56nn   N=0  ZC   der Speicherzelle nn+X
NOP         EA             wartet
ORA $nnnn   0Dnnnn N    Z  ODER-Verkn. Zelle nnnn mit Akku
ORA $nn     05nn   N    Z  ODERd Speicherzelle nn
ORA #$nn    09nn   N    Z  ODERd Byte nn mit Akku
ORA $nnnn,X 1Dnnnn N    Z  ODERd Speicherzelle nnnn+X
ORA $nnnn,Y 19nnnn N    Z  ODERd Speicherzelle nnnn+X
ORA $nn,X   15nn   N    Z  ODERd Speicherzelle nn+X
ORA ($nn,X) 01nn   N    Z  ODERd Byte aus (nn+X)
ORA ($nn),Y 11nn   N    Z  ODERd Byte aus (nn)+Y
PHA         48             bringt Akku auf Stapel
PHP         08             bringt Status auf Stapel
PLA         68     N    Z  holt Akku vom Stapel
PLP         28     NVBDIZC holt Status vom Stapel
ROL A       2A     N    ZC rotiert den Akku nach links
ROL $nnnn   2Ennnn N    ZC   die Speicherzelle nnnn
ROL $nn     26nn   N    ZC   die Speicherzelle nn
ROL $nnnn,X 3Ennnn N    ZC   die Speicherzelle nnnn+X
ROL $nn,X   36nn   N    ZC   die Speicherzelle nn+X
```

ROR A	6A	N ZC	rotiert den Akku nach rechts
ROR $nnnn	6Ennnn	N ZC	die Speicherzelle nnnn
ROR $nn	66nn	N ZC	die Speicherzelle nn
ROR $nnnn,X	7Ennnn	N ZC	die Speicherzelle nnnn+X
ROR $nn,X	76nn	N ZC	die Speicherzelle nn+X
RTI	40		Rückkehr aus Interrupt
RTS	60		Rückkehr aus Unterprogramm
SBC $nnnn	EDnnnn	NV ZC	sub. Speicherzelle nnnn zum Akku
SBC $nn	E5nn	NV ZC	subtra. Speicherzelle nn
SBC #$nn	E9nn	NV ZC	subtra. Byte nn zum Akku
SBC $nnnn,X	FDnnnn	NV ZC	sub. Speicherzelle nnnn+X
SBC $nnnn,Y	F9nnnn	NV ZC	sub. Speicherzelle nnnn+X
SBC $nn,X	F5nn	NV ZC	sub. Speicherzelle nn+X
SBC ($nn,X)	E1nn	NV ZC	sub. Byte aus (nn+X)
SBC ($nn),Y	F1nn	NV ZC	sub. Byte aus (nn)+Y
SEC	38	C=1	setzt C
SED	F8	D=1	setzt D
SEI	78	I=1	setzt I
STA $nnnn	8Dnnnn		speichert Akku in Zelle nnnn
STA $nn	85nn		speichert Akku in Zelle nn
STA $nnnn,X	9Dnnnn		speichert Akku in Zelle nnnn+X
STA $nnnn,Y	99nnnn		speichert Akku in Zelle nnnn+Y
STA $nn,X	95nn		speichert Akku in Zelle nn+X
STA ($nn,X)	81nn		speichert Akku in Zelle (nn+X)
STA ($nn),Y	91nn		speichert Akku in Zelle (nn)+Y
STX $nnnn	8Ennnn		speichert X in Zelle nnnn
STX $nn	86nn		speichert X in Zelle nn
STX $nn,Y	96nn		speichert X in Zelle (nn)+Y
STY $nnnn	8Cnnnn		speichert Y in Zelle nnnn
STY $nn	84nn		speichert Y in Zelle nn
STY $nn,X	94nn		speichert Y in Zelle (nn)+X
TAX	AA	N Z	bringt Akku nach X
TAY	A8	N Z	bringt Akku nach Y
TSX	BA	N Z	bringt Status nach X
TXA	8A	N Z	bringt X in Akku
TXS	9A	NVBDIZC	bringt X in Status
TYA	98	N Z	bringt Y in Akku

III. Speicherbelegungsplan

In Klammern sind die Kapitel angegeben, in denen die Benutzung der Speicherzelle erklärt wird.

0	Prozessorport Datenrichtungsregister
1	Prozessorport Datenregister (3.2.)
2	unbenutzt
3 / 4	Vektor f. Umwandlung Fließkomma - Fest
5 / 6	Vektor f. Umwandlung Fest - Fließkomma
7	Suchzeichen
8	Flag f. Sonderzeichenmodus
9	TAB-Spalte
10	0= LOAD bzw. 1= VERIFY letzter Befehl
11	Zeiger f. Eingabepuffer / Dimensionen
12	DIM-Flag
13	Variablentyp: FF=String, 00=Zahl
14	80= Integervar., 00= Fließkommavar.
15	Sonderzeichenmodus bei LIST
16	Flag für FNx
17	letzter Zuweisungstyp: 0= INPUT, 64= GET 152= READ, gibt an, woher die letzte Zuw, kam
18	Vorzeichen bei ARCTAN / letzter Vergleich: 1= größer, 2= "=", 4= kleiner
19	aktueller File POKE 19,64 bewirkt INPUT ohne ?
20 / 21	Integer-Zahl, z.B. Adressen, FRE(0)
22	Vektor f. Stringstack (12.9.)
23 / 24	Zeiger auf letzten String POKE 24,0 behebt Blockierung nach FORMULAR TOO COMPLEX-ERROR
25 - 33	Stringstack
34 - 37	diverse Zeiger
38 - 42	Arithmetikregister
43 / 44	Zeiger auf BASIC-Programm-Anfang (3.3., 4.2.)
45 / 46	Zeiger auf Variablenstart (3.3., 4.2.)
47 / 48	Zeiger auf Beginn der Felder
49 / 50	Zeiger auf Ende der Felder

51 / 52	Zeiger auf Stringanfang (bewegt sich abwärts)
53 / 54	Stringhilfszeiger
55 / 56	Zeiger auf Speichergrenze (3.3.)
57 / 58	augenblickliche BASIC-Zeilennummer
59 / 60	vorherige BASIC-Zeilennummer (12.6.)
61 / 62	Zeiger auf nächsten Befehl für CONT (12.2.) Wollen Sie ein Programm nach einem Error mit CONT weiterlaufen lassen, POKEn Sie hier einfach die Zeilennumer ein, bei der es weitergehen soll
63 / 64	aktuelle DATA-Zeile (12.5.)
65 / 66	Zeiger auf nächstes DATA-Element (12.5.)
67 / 68	Zeiger auf letztes DATA/INPUT/GET
69 / 70	aktuelle Variable (2 Buchstaben) REAL: 2 ASCII-Codes, INTEGER: beide Codes um 128 erhöht, STRING: 2. Code um 128 erhöht
71 / 72	Zeiger auf aktuelle Variablen
73 / 74	Zeiger auf aktuelle FOR-NEXT-Variable
75 / 76	Hilfsregister f. BASIC-Programmzeiger
77	Hilfsregister f. Vergleiche
78 / 79	Zeiger für FNx
80 - 83	Hilfsregister f. Strings
84 - 86	Sprungvektor für Funktionen
87 - 91	Arithmetik-Akku 3
92 - 96	Arithmetik-Akku 4
97 - 101	Arithmetik-Akku 1
102	Vorzeichen von Akku 1
103	Zähler f. Polynomauswertung
104	Rundungsbyte für Akku 1
105 - 109	Arithmetik-Akku 2
110	Vorzeichen von Akku 2
111	Vergleichsregister Akku 1 & 2
112	Rundungsbyte
113 - 114	Zeiger f. Polynomauswertung
115 - 138	CHRGET-Routine holt nächstes Byte aus BASIC-Pgm
122 / 123	BASIC-Programmzeiger
139 - 143	letzter RND-Wert
144	Status (wie Variable ST)
145	Flags f. Tastaturspalte 1 (10.1.)

146	Zeitkonstante f. Cassettenbetrieb
147	0= LOAD, 1= VERIFY (4.4.)
148	Flag für IEC-Bus
149	Zeichen für IEC-Bus
150	Flag f. End of Tape (Kassettenende) (4.4.)
151	Zwischenspeicher f. Register
152	Anzahl der geöffneten Files (4.4.)
153	aktuelles Eingabegerät (normal: 0) (4.4.)
154	aktuelles Ausgabegerät (CMD, norm.: 3)(4.4.)
155	Paritätsbyte bei Kassettenbetrieb
156	Flag für Byte empfangen
157	Ausgabemodus 0 normal, 64 zusätzlich Betriebssystemmeld. 128 zus. Direktmodusmeldungen
158	Prüfsumme bei Kassettenbetrieb
159	Fehlerkorrektur bei Kassettenbetrieb
160 - 162	Uhr
163	Bitzähler bei serieller Ausgabe
164	Zähler bei Bandbetrieb
165	Zähler für Schreiben auf Band
166	Zeiger in Kassettenpuffer
167 - 171	Flags für Bandbetrieb
172 / 173	Zeiger für Kassettenpuffer
174 / 175	Zeiger auf Programmende (LOAD / SAVE) (4.1.)
176 - 177	Zeitkonstanten für Kassette
178 / 179	Zeiger auf Kassettenpufferstart
180	Bitzähler (Kassette)
181	Nächstes Bit für RS 232 (Senden)
182	Auszugebendes Byte
183	Zeichenanzahl im Filenamen (4.1.)
184	aktuelle logische Filenummer (4.4.)
185	aktuelle Sekundäradresse (4.4.)
186	aktuelle Gerätenummer (4.4., 4.1.)
187 / 188	Zeiger auf Filennamen (4.4., 4.1.)
189	Hilfsregister f. serielle Ausgabe
190	Blockzähler f. Bandbetrieb
191	Wortpuffer f. serielle Ausgabe
192	Flag f. Kassettenmotor (0 = aus, 1 = an)
193 / 194	Startadresse f. LOAD / SAVE (4.1.)

195 / 196	Endadresse f. LOAD / SAVE
197	gedrückte Taste
198	Anzahl Tastendrücke im Puffer (12.1., 9.5.)
199	Flag für RVS (5.6.)
200	Zeilenlänge bei Eingabe (Zeiger)
201 / 202	Zeiger auf Eingabecursor (Zeile, Spalte)
203	gedrückte Taste (9.5.)
204	Flag f. Cursor (0= blinkt) (5.6.)
205	Zähler für Blinkzeit
206	Zeichen unter Cursor
207	Blinkflag (5.6.)
208	Flag f. Eingabe von Tastatur / Bildschirm
209 / 210	Zeiger auf aktuelle Bildschirmzeile (5.6.)
211	Cursorspalte (5.6.)
212	Art des Cursors (programmiert / direkt)
213	Länge der Bildschirmzeile (40 / 80)
214	Cursorzeile (5.6.)
215	letzte Taste
216	Anzahl der Inserts (5.6.)
217 - 242	Highbytes der Zeilenanfänge
243 / 244	Cursorposition im Farb-RAM (5.6.)
245 / 246	Zeiger auf Tastaturdekodiertabelle
247 / 248	Zeiger auf Eingabepuffer für RS 232
249 / 250	Zeiger auf Ausgabepuffer für RS 232
251 - 254	Freie Bytes für Betriebssystem
255	Beginn des BASIC-Speichers (* 64)
256 - 511	Prozessorstack
256 - 266	Zwischenspeicher für Formatumwandlung
256 - 318	Korrektur von Bandfehlern
512 - 600	BASIC-Eingabepuffer
601 - 610	logische Filenummern
611 - 620	Gerätenummern
621 - 630	Sekundäradressen
631 - 640	Tastaturpuffer (12.1.)
641 / 642	Zeiger auf BASIC-RAM-Start
643 / 643	Zeiger auf BASIC-RAM-Ende
645	Flag f. Zeitfehler auf seriellem Bus
646	aktuelle Schriftfarbe (5.6.)
647	Farbe unter Cursorposition (5.6.)
648	Highbyte der TV-RAM-Basisadresse (5.5.)

649	max. Länge des Tastaturpuffers (9.3.)
650	Flag f. Repeat (0=norm., 128=alle, 127=aus) (9.4.)
651	Zähler f. Repeatgeschwindigkeit (9.4.)
652	Zähler f. Repeatverzögerung (9.4.)
653	Flag f. SHIFT, Commodore und Control (9.2.)
654	wie 653
655 / 656	Zeiger auf Tastaturdekodiertabelle (9.3.)
657	Flag f. Zeichensatzumschaltungssperre
658	Flag für Scrolling
659	Kontrollregister f. RS 232
660	Befehlsregister f. RS 232
661 / 662	Bit-Zeit
663	Statusregister f. RS 232
664	Anzahl zu sendender Bits f. RS 232
665 / 666	Baudrate für RS 232
667	Zeiger auf empfangenes Byte f. RS 232
668	Zeiger auf Eingabe von RS 232
669	Zeiger auf auszugebendes Byte f. RS 232
670	Zeiger auf Ausgabe aus RS 232
671 / 672	Zwischenspeicher f. IRQ bei Bandbetrieb
673	NMI-Flag CIA 2
674	Timer A des CIA 1
675	Interruptflag des CIA 1
676	Flag f. Timer A
677	Bildschirmzeile
678 - 767	freier RAM-Bereich
704 - 766	Sprite-Block 11
768 / 769	Zeiger f. Fehlermeldung
770 / 771	Zeiger auf BASIC-Warmstart
772 / 773	Zeiger auf Umwandlung Text - Kode
774 / 775	Zeiger auf Umwandlung Kode - Text
776 / 777	Zeiger auf Befehlsausführung
778 / 779	Zeiger auf Ausdrucksauswertung
780	Akku für SYS Bei einem SYS-Befehl wird in diesem und den folgenden Registern angegeben, welche Werte die Prozessorregister haben sollen. Nach der Ausführung werden hier die Ergebnisse zurückgemeldet.

781	X-Register für SYS
782	Y-Register für SYS
783	P-Register für SYS
784 - 787	USR-Sprung (Adresse in 785 / 786)
788 / 789	Zeiger auf Hardware-Interrupt (9.3., 9.2.)
790 / 791	Zeiger auf BRK-Interrupt
792 / 793	Zeiger auf NMI (9.3.)
794 / 795	Zeiger auf OPEN
796 / 797	Zeiger auf CLOSE
798 / 799	Zeiger auf Zeicheneingabe
800 / 801	Zeiger auf Zeichenausgabe (12.6.)
802 / 803	Zeiger auf Kanäle löschen (12.6.)
804 / 805	Zeiger auf Eingabe
806 / 807	Zeiger auf Ausgabe
808 / 809	Zeiger auf STOP-TASTE abfragen (9.3.)
810 / 811	Zeiger auf GET
812 / 813	Zeiger auf alle Kanäle schließen
814 / 815	Zeiger auf Benutzer-IRQ
816 / 817	Zeiger auf LOAD
818 / 819	Zeiger auf SAVE (12.6.)
820 - 827	freier RAM-Bereich
828 - 1019	Kassettenpuffer
832 - 894	Sprite-Block 13
896 - 958	Sprite-Block 14
960 - 1022	Sprite-Block 15
1023	frei
1024 - 2023	TV-RAM
2024 - 2039	frei
2040 - 2047	Zeiger für Sprites
2048 - 40960	BASIC-Speicher
8192 - 16192	Bit-Map f. hochauflösende Grafiken
40960 - 49151	BASIC-Interpreter-ROM
42039	POKE 781,x: SYS 42039 gibt x-ten Fehler aus
42115	End ohne Ready (12.6.)
43121	SYS 43121 = RUN 0
44808	Syntax Error (12.6.)
49152 - 53247	4 K RAM für Maschinenprogramme
53248 - 57343	Charaktergenerator (5.3.)
53248 - 53294	Register des VIC
53267 / 53268	Lightpen-Register (10.3.)

53295 - 54271	977 Bytes leer
54272 - 54300	Register des SID
54297 / 54298	Paddles (10.2.)
54301 - 55295	995 Bytes leer
55296 - 56295	Color-RAM
56296 - 56319	24 Bytes leer
56320 - 56335	Register des CIA 1
56320 / 56321	Tastaturabfrage und Joysticks (9.2., 10.)
56322	Tastatursteuerung (10.1.)
56325	Repeatgeschwindigkeit
56334	Interrupt-Register (1.2., 11.1.2.)
56336 - 56575	240 Bytes leer
56576 - 56591	Register des CIA 2
56577 & 56579	USER-PORT-Register (11.2.)
56592 - 57343	752 Bytes leer
57334 - 65535	Betriebssystem-ROM
58253	Einschaltbild (12.6.)
58732	Cursor setzen (5.6.)
59903	POKE 781,x: SYS 59903 löscht x-te Zeile
62954	SAVE-Routine (4.1.)
64931	SYS 64931 bewirkt RESET v. CIAs u. SID-Lautst.
65499	TI=0 (12.6.)

IV. VIC und SID

VIC: Basisadresse: 53248

Reg.	Inhalt
0-15:	Sprite-Koordinaten (X,Y)
16:	MSB der Y-Koordinaten
17:	Steuerregister (5.3., 5.6., 6.3.)
18:	Rasterzeile f. Interrupt
19/20:	Lightpen-Position (X,Y)
21:	Sprite-Enable
22:	Steuerregister (5.3., 6.3.)
23:	Y-Ausdehnung
24:	Video-RAM-/Char-ROM-Steuerregister
25/26:	Interruptsteuerung (5.4., 5.5., 6.2., 6.3.)
27:	Sprite-Priorität (7.3.)
28:	Sprite-Multicolor-Modus (7.1.)
29:	X-Ausdehnung
30:	Sprite-Sprite-Kollision (7.2.)
31:	Sprite-Daten-kollision (7.2.)
32:	Rahmenfarbe
33-36:	Hintergrundfarben 0 - 3 (5.3.)
37/38:	Multi-Color-Farben f. Sprites (7.1.)
39-46:	Spritefarben

SID: Basisadresse 54272 (8.2.)

Reg.	Inhalt
0/1:	Frequenz (für andere Stimmen 7/8, 14/15)
2/3:	Pulsbreite (auch 9/10 und 16/17)
4:	Steuerregister (auch 11 und 18)
5:	Anschlag und Abschwellzeit (auch 12 und 19)
6:	Halten und Ausklingen (auch 13 und 20)
21/22:	Filterfrequenz
23:	Filterkontrollregister
24:	Lautstärke und Filterauswahl
25/26:	Paddles (10.2.)
27:	Oszillator
28:	Hüllkurve Stimme 3

V. Stichwortverzeichnis

L

M

N

O

P

R

S

Z

DAS STEHT DRIN:

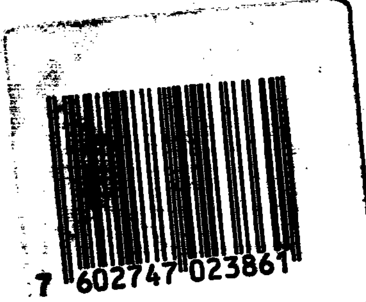
Dieses Buch erklärt Ihnen leichtverständlich den Umgang mit Peeks & Pokes. Es enthält eine Beschreibung aller nutzbaren Speicheradressen und führt in die Hardware Ihres C 64 ein. Die vielen sofort einsetzbaren Programme haben schnell dafür gesorgt, daß dieses Buch schon in der ersten Auflage zu einem unverzichtbaren Nachschlagewerk für jeden interessierten Programmierer wurde.

Aus dem Inhalt:

- Die Arbeitsweise der CPU
- Was ist ein Betriebssystem
- Wie arbeitet der BASIC-Interpreter
- Beschreibung und Nutzung der Zeropage
- Pointer & Stacks
- Speicherbelegungsplan
- Massenspeicherung & Peripherie
- Die Spriteregister
- Programmierung der Schnittstellen
- Interruptprogrammierung
- Leichtverständliche Einführung in die Maschinensprache

UND GESCHRIEBEN HAT DIESES BUCH:

Hans Joachim Liesert, Informatikstudent an der RWTH-Aachen, ist langjähriger Computerfreak und hat unter anderem die PEEKS & POKES-Bücher zum C 128 und Schneider CPC geschrieben.



32-0